

"Modular Regression Testing": Connections to Component-Based Software

Bruce W. Weide

Dept. of Computer and Information Science
The Ohio State University
2015 Neil Ave.
Columbus, OH 43210 USA
+1 614 292 1517
weide.1@osu.edu

ABSTRACT

Many have argued that software that is not designed to support modular reasoning about its behavior is inherently fragile and costly to maintain, and that software engineers should seek to achieve the modular reasoning property to help overcome these problems. But some people resist these claims, taking one of two contradictory positions:

1. Modular reasoning is inherently limited to impractical purely functional programs where there is no state and there are no side-effects.
2. Modular reasoning is possible for any reasonably "well-designed" software system written in a modern imperative object-oriented language that uses its sophisticated encapsulation mechanisms.

Explanations of why (1) is wrong have been relatively effective. We suspect this is because both experimental and (more recently) commercial software has been built in C++ in a disciplined way that supports modular reasoning about its behavior, and it has (among other advantages) dramatically lower defect rates than "normal" software of like kind.

Explanations of why (2) is wrong have been less effective. We suspect this is because they have been based on synthetic examples that appear to be pathological and therefore of little practical consequence. Using a thought experiment involving regression testing of systems having features that no one should doubt are just like "real" software, we make another stab at giving a convincing argument on this point.

Keywords

Component-based software, compositional reasoning, modular reasoning, regression testing, testing.

BACKGROUND

In other papers [3, 4, 8, 9, 10] we have argued that:

- Software must be, and can be, designed to support the modular reasoning property (a.k.a. "compositional reasoning" about its behavior, or "local certifiability" of correctness) in order to be economically maintainable.
- There are plenty of easily avoidable design flaws that thwart modular reasoning about behavior in most existing software.

A system has the *modular reasoning property* if it is possible to reason soundly about the behavior of any component in it — in other words, to predict its behavior and/or to check whether its behavior satisfies a specification — by consulting only the source code for that component and the specifications of the behaviors of the components it uses. Crucially, reasoning about a component's behavior must *not* involve consulting the source code for the other components it uses, nor knowing how the component is being used by other components. Design-by-contract is the underlying principle that informs the *design* of such a system. Conversely, the ultimate success of design-by-contract depends on the system *implementation* actually satisfying the modular reasoning property.

Unfortunately, the modular reasoning property does not hold unless you are careful and disciplined when designing components for, and coding their implementations in, modern imperative and object-oriented languages [1, 4, 8, 9]. Our call therefore has been to pay close attention to "microarchitectural" issues [3] when designing, specifying, and implementing software components and systems built from them. If software is poorly designed at this fundamental level, then no amount of tool support or managerial savvy can make that software economically maintainable. For many years we based the case for discipline on two complementary arguments: first principles about good design, and observations about the difficulty of reverse-engineering poorly-designed software. Recently we have been able to show empirical benefits for commercial software [4].

POSITION

It is foolishly optimistic to suggest that the arguments in support of modular reasoning have had *any* notable impact on software engineering practice. As witnesses we call all the modern component libraries used by "real" programmers: the STL for C++, Java libraries, etc. We're still trying to explain why the modular reasoning property is important, how it can be rather painlessly achieved by observing a little discipline, and why traditionally-designed software doesn't have it.

It is this last point that now seems to be the weakest link in terms of explanation. So the question considered in this paper is whether relating modular reasoning to the problem of regression testing of software might provide new insights that could help more people understand what we've been talking about. Why doesn't most "real" software have the modular reasoning property, and how bad a problem is this? Does it really impact software engineering practice? I defend the following proposition:

Careful consideration of the problem of regression testing of component-based software, comparing situations where the modular reasoning property does and does not hold, can shed considerable light on both the importance and pervasiveness of the property and on the kinds of software design and implementation flaws that thwart it.

My approach to defending this thesis is a bit unusual. I describe a thought experiment (although one that could be carried out, in principle) involving regression testing and use it to discuss challenges for the program testing and analysis community, the metrics community, advocates of reverse engineering, and others. It would be easy to misunderstand the purpose of the paper — to imagine that it is advocating an "approach" to program testing and analysis, metrics, or reverse engineering of C++ programs. It is actually about explaining to non-believers both the practical importance and the fragility of the modular reasoning property. There are several related but technically tangential issues that might be of independent interest to the CBSE4 participants. I will be pleased to discuss these at the workshop should the occasion arise.

- Design-by-contract behavioral specifications and their use in formal modular reasoning by verification [1, 9].
- Subtle threats to modular reasoning that arise from using more sophisticated language features (e.g., pointers, inheritance, templates) that are not needed to prove the point and are therefore not included in thought experiment.
- How modular reasoning can be achieved by disciplined use of modern imperative and object-oriented languages such as C++, even in the presence of objects with state, methods with side-effects, etc. [4].

MODULAR REGRESSION TESTING

An important issue in software testing is regression testing, which involves repeating the execution of a test suite after software has been changed, in an attempt to reveal defects introduced by the change. One reason this is important is that it is often very expensive. If the test suite is comprehensive then it can take significant time and resources to conduct and evaluate the new test. Some commercial software developers, therefore, routinely do regression testing after even minor changes, and it is not unusual for each regression testing session to run overnight (if it is automated) or even longer (if human attention is required to decide whether the software is responding appropriately).

Component-based software seems to offer a way to address this problem because a common kind of change in well-designed component-based systems is for one component to replace another. Even in systems that are not consciously designed for plug-compatibility, if changes are incremental and a group of changes is limited to a single component, the effect is *as if* one component has replaced another. Is it possible to confine regression testing of software that has undergone component-level or component-scoped maintenance to the "vicinity" of the replaced or modified component? Or does the entire system have to be tested again?

Assume you have a software system written in C++, and its "components" are simply C++ functions (which in C++ may actually be procedures, i.e., operations with side-effects on their arguments and/or global or static data). Assume there are no classes, templates, inheritance, etc. — not even pointers. These assumptions are convenient for this paper because it is easy to relate the scenario to actual software most people have dealt with, and because they dramatically simplify the arguments yet still allow the main point to show through.

A diagram of part of such a software system is shown in Figure 1. Nodes represent C++ functions and arcs represent calling relationships; I explain circles around nodes later. So, for example, P calls Q and R; B calls R; etc.

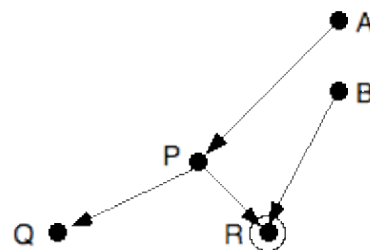


Figure 1

Now consider what happens if you change this software system by replacing P by P', leaving the situation illustrated in Figure 2. You might have made this substitution because, for example, P' (allegedly) computes the same thing

as P, but does it a lot faster by calling R, S, and T than by calling Q and R.

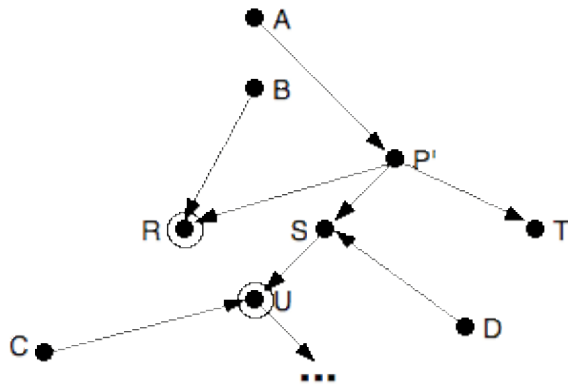


Figure 2

In an ideal world you should not have to regression test the whole system just because of this minor maintenance, any more than a mechanic should have to road-test your car on the test track after changing the oil. In a moment of optimism, you might hope to use this method instead:

- Log every call to P while testing the software system the first time, recording the values of P's arguments and any global data P refers to, upon each call and upon each corresponding return. (Ignore the questions of how you would know to do this before deciding to replace P, how to record values that are not of built-in types such as int, how much this would cost, etc.)
- "Play back" the calls to P into a simple test driver for P', comparing the results produced by P' to the recorded results produced by P for the same series of calls. (Ignore the question of whether P' still might be substitutable for P even if it did not give exactly the same results, because you want to be conservative and are willing to pay for a complete traditional regression test if there is any doubt.)

The question is whether this approach, which we call **modular regression testing**, is effective in the following sense: If P' actually gives the same results as P for the logged calls, could a complete traditional regression test of the entire system possibly reveal any defects?

SOUNDNESS

If a complete traditional regression test *could not* reveal other defects, then modular regression testing is **sound** and could safely be used to achieve the same degree of confidence as a complete regression test — and it might turn out to be far less expensive. If a complete traditional regression test *could* reveal other defects, then you would need a complete test anyway in order to gain the same degree of confidence, and modular regression testing is simply a bogus concept.

Quick, before reading on... which is it?

Factors Affecting Soundness

After some reflection, it should be clear that there are certain conditions under which modular regression testing is sound, and other conditions under which it is not. This short paper can't get into all of them, but it is relatively easy to motivate the issues involved by considering one specific issue that affects soundness.

Let's call a component **clean** if its observable behavior does not depend on any "static" data values kept privately within the component. If a component is not clean, let's call it **dirty**. (I don't mind the value judgments implied by this choice of terminology.) In Figures 1 and 2, dirty components are shown inside circles. For example, R might store and retrieve some information using a private temporary file that outlives a given call. Or, U's observable behavior might depend on a static variable recording the number of times U has been invoked.

The issue here is not whether P and P' themselves are clean or dirty, but whether the components they rely on are clean or dirty. Both P and P' call R, which is dirty. These calls cause no problem if P and P' give identical answers to their callers. But what about other calls to R from elsewhere in the system, e.g., from B? Similarly, the indirect calls from P' to U (through S) might affect the outcomes of other calls to U, e.g., from C.

The "Expanding Module Phenomenon"

The above problems might be surmountable, if you expand the hypothetical "module boundary" through which calls are logged so the interior includes more than just P itself. You can record all calls to P (P') that cross the expanded module boundary, e.g., from A; all calls across the boundary to the dirty component R, e.g., from B; all calls across the boundary to the dirty component U, e.g., from C; and all calls across the boundary to the clean component S whose behavior depends on that of the dirty component U, e.g., calls from D.

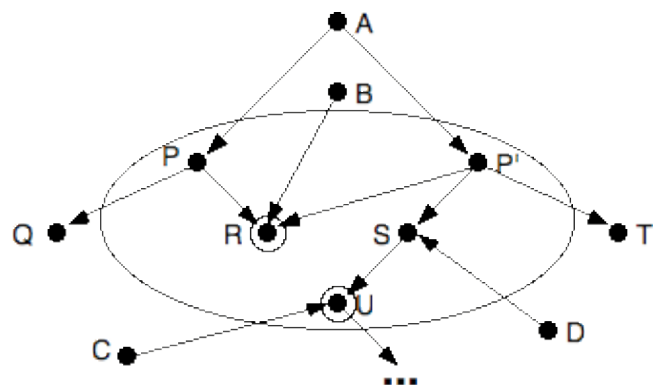


Figure 3

The expanded module boundary is defined by constructing all call paths starting from P and P', and including within the boundary all components on each such path through the

last dirty component on that path. This subassembly of (actual) components then becomes a (mythical, from the standpoint of the actual source code) new "component". In other words, P' can't be regression-tested alone; you have to regression-test the entire expanded module induced by the proposed substitution of P' for P .

Still, this might involve far less work than regression-testing the entire system. But would it really be less work? How big is the expanded module for typical "real" software maintenance activities?

We have claimed [10] that, for real programs, the modular reasoning property frequently does not hold for a given component because there are quite a few components that are dirty in the sense explained above, or in other senses (e.g., they use visible pointers with potential aliases; reference semantics; implementation inheritance). That is, we have claimed that if you try to reverse engineer a typical software system, you will be stymied because you can't tell from information in the individual components, e.g., P and P' , whether it is possible to substitute P' for P without breaking something in the far reaches of the system. Therefore, you need to look at potentially *the entire system* to check for possible weird interactions.

The validity of this claim is an empirical question, as it says something about how software engineers actually design and implement systems. The idea of modular regression testing suggests one way to test the claim, and even to measure it. With sophisticated program analysis tools [2] it might be possible to construct the expanded module for any proposed component-level substitution. Our earlier claims about the difficulty of reverse engineering are therefore falsifiable: someone just needs to carry out the modular regression testing thought experiment in an empirical study of actual software systems in an attempt to show that the expanding module phenomenon is not severe for "real" software.

Note that software that is intentionally designed to support the modular reasoning property simply does not suffer from the expanding module phenomenon at all.

RELATED WORK

There has been plenty of interesting work on regression testing, including suggestions for how to use static program analysis to limit the test cases that must be involved in regression testing [5, 6, 7]. But, to my knowledge, the proposed modular regression testing technique has not been previously suggested. It would limit not the test cases but the parts of the software that would be involved in regression testing. Perhaps this apparent novelty arises because it is clear to those in the testing community that the very idea of modular regression testing of normal software is doomed to fail. Why? Because our claim about the intractability of reverse engineering [10] is essentially correct — and this is tantamount to the unsoundness of modular regression test-

ing as outlined above. If modular regression testing actually were sound for "real" software, there would be only unit testing; it would be pointless to waste effort doing integration testing and system testing, or equivalently, it would be pointless to consider them as different than simply unit testing of bigger and bigger units.

The point is that even if you don't actually try to carry out modular regression testing in practice, the idea seems to shed some light on determining how well or how poorly a given software system is structured from the standpoint of the modular reasoning property. I am pleased to offer it as the basis for defining a new software metric.

ACKNOWLEDGMENTS

Joe Hollingsworth and Murali Sitaraman are at least as responsible as I am for the ideas presented here. They should, of course, not be blamed for any errors that have survived our discussions and made it into this paper, or for flaws in the presentation.

An earlier version of this paper was accepted to the 9th Workshop on Software Reuse, but I was unable to attend the workshop to discuss it. Many thanks to Judith Stafford for agreeing to reconsider it for the CBSE4 Workshop.

Finally, the three anonymous reviewers made many useful comments that helped to focus the presentation on the points I really wanted to make.

This work has been supported by the National Science Foundation under grants DUE-9555062, CDA-9634425, and CCR-0081596, by the Fund for the Improvement of Post-Secondary Education under project number P116B60717, by Microsoft Research, and by Lucent Technologies. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Science Foundation, the U.S. Department of Education, Microsoft, or Lucent.

REFERENCES

1. Cook, S.A. Soundness and completeness of an axiom system for program verification. *SIAM J. Comp.* 7, 1 (1978), 70-90.
2. Harrold, M.J., Larsen, L., Lloyd, J., Nedved, D., Page, M., Rothermel, G., Singh, M., and Smith, M. Aristotle: a system for the development of program-analysis-based tools. In *Proceedings of the 33rd ACM Annual Southeast Conference*, ACM, 1995, 110-119.
3. Hollingsworth, J.E., and Weide, B.W. One architecture does not fit all: micro-architecture is as important as macro-architecture. In *Proceedings 7th Annual Workshop on Software Reuse* (St. Charles, IL, August 1995), 5 pp. On line at:

<ftp://gandalf.umcs.maine.edu/pub/WISR/wisr7>

4. Hollingsworth, J.E., Blankenship, L., and Weide, B.W. Experience report: using RESOLVE/C++ for commercial software. In *Proceedings ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering* (San Diego, CA, November 2000), ACM, 11-19.
5. Rothermel, G., and Harrold, M.J. Experience with regression test selection. *Empirical Software Engineering Journal* 2, 2 (1997), 78-187.
6. Rothermel, G., and Harrold, M.J. A safe, efficient algorithm for regression test selection. *ACM Transactions on Software Engineering and Methodology* 6, 2 (1997), 173-210.
7. Rothermel, G., and Harrold, M.J. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering* 24, 6 (1998), 401-419.
8. Weide, B.W., and Hollingsworth, J.E. Scalability of reuse technology to large systems requires local certifiability. In *Proceedings 5th Annual Workshop on Software Reuse* (Palo Alto, CA, October 1992), 7 pp. On line at:

<ftp://gandalf.umcs.maine.edu/pub/WISR/wisr5>
9. Weide, B.W., Heym, W.D., and Ogden, W.F. Procedure calls and local certifiability of component correctness. In *Proceedings 6th Annual Workshop on Software Reuse* (Owego, NY, October 1993), 5 pp. On line at:

<ftp://gandalf.umcs.maine.edu/pub/WISR/wisr6>
10. Weide, B.W., Heym, W.D., and Hollingsworth, J.E. Reverse engineering of legacy code exposed. In *Proceedings 17th International Conference on Software Engineering* (Seattle, WA, April 1995), ACM, 327-331.

