Thoughts on Applicability

Titus Winters[1,]

[a]*Adobe, Inc, 104 5th Ave, 4th Floor, 10011, New York, New York, USA*

1

1. Introduction

For the past decade, I have tried to reduce the gap between industry and academia, attending conferences, describing industry approaches to tasks like refactoring and testing, and leading the work on the book "Software Engineering at Google." Even though I'm more focused on theory than most industry practitioners, attending ICSE or reading JSS is often frustrating. In many cases, talks and papers seem irrelevant in practice even in these top-tier venues. Thus, when the JSS EICs suggested writing about this applicability gap, I felt I had to jump at the opportunity. I've spent the past week going over my notes from ICSE'23 and reading a selection of recent open-access articles from JSS. What follows here is a hopefully-constructive critique of the type of work presented in these forums, followed by a selection of topics and research questions that I would love to see covered in future research. My hope in writing this article is to contribute to the badly needed discourse between research and practice, to help bridge these domains. I certainly do not wish to single out anyone or cast aspersions - after all, these are talks and papers that made it into top-tier venues!

With that said, let us first look at the applicability concerns faced by academic software engineering papers.

2. Solutions Without Clear or Practical Context

At ICSE this year I remember at least one presentation on algorithms to connect "self admitted technical debt" (a.k.a. "TODO" comments) to issues in an issue tracker. The researchers were deploying machine learning techniques to automate this process, and evaluating what aspects of feature selection (comments, authorship, surrounding code, pull request metadata, etc) were most relevant.

[1]To appear in JSS: https://doi.org/10.1016/j.jss.2024.112086

Coincidentally, at roughly the same time, I got into discussions about Google style standards for TODO statements[2]. Google has had a consistent internal format for many years, preferring `TODO(link to bug) - Description of issue` for new code. The new discussion focused on compatibility with external tools and standards. Apparently the default syntax highlighting in popular source browsers (GitHub) and some editors (XCode), assumes `TODO: link to bug - Description`. We reasoned that tools will eventually converge, there's no reason to wait, and made the change. Personally, I took it as a given that any company or OSS project that cares will have mandated a similar format.

Thus, when presented with precision/recall graphs for automatically matching SATD to issues, I have to ask: isn't there an easier way? Why ask an algorithm to do a poor job of a task that a human can pretty obviously do cheaply? Who would such an algorithm be for? If any team or organization needed that matching, it would be much easier to tell developers to leave that sort of paper trail in their TODOs. With more than a decade of involvement in Google style discussions, I have only once heard anyone argue that such a requirement is a problem[3].

For me, this is a frequent concern for academic software engineering papers: what kind of constraints are needed to motivate the approach being researched? I have to imagine that if we were clearer about the assumed constraints, it would be easier to identify the most relevant and most impactful research. Not coincidentally, this would also make it easier for practitioners to identify what is relevant to them. It could even motivate practitioners to re-evaluate their own constraints, in order to adopt a valuable tool or practice that is inapplicable in their current environment!

I highly recommend researchers consider what assumptions are needed to motivate the problems they are investigating. Are we assuming developer behavior cannot be changed? Or that tests exist? Or that tests don't exist? Are we assuming code cannot be shared? Or cannot be rebuilt? These constraints absolutely dominate the applicability of academic research in industry domains: too few constraints and the results aren't applicable, too many and the results aren't meaningful. Be clear about those assumptions. If strange assumptions seem to be making the problem hard, try an easier approach[4].

[2] I was still employed at Google at the time

[3] That was specifically in the discussion of disallowing TODO(username), with the notion that some teams have many small and short-lived TODOs, and that use of an issue tracker for that volume would be annoying. Even the primary author of that argument admitted it was not a particularly critical debate.

[4] If we don't know what assumptions are reasonable or not, surveys of the practice to identify norms seem like a very valuable place to start! Or feel free to email me, I'm happy to share what perspective I have.

3. Answering Research Questions that Nobody is Asking

A recent Requirements Engineering paper set out to model which prioritization criteria were most used at different phases of the software workflow, and to discern whether those criteria change based on how far along a given feature has progressed.

This certainly does not suffer from the same issues as my previous example. Countless discussions and meetings happen on a daily basis, as teams decide what to tackle next. Requirements Engineering and prioritization decisions are quite relevant. My concern with work like this is not that the problem isn't legitimate, it's that I don't know what to do with any answers to the research questions.

Imagine a once-in-a-lifetime miracle occurs and we discover an absolutely perfect algorithm to describe dynamic prioritization, taking into account all available data in our issue trackers. We validate this against not only the original issue-tracking dataset but thousands of novel datasets, and the algorithm almost perfectly matches behavior in the wild - only differing where human project managers or team leads made "non-optimal" priority decisions.

Even with such a result ... what do we expect to be different in practice? Is the intent to automate away project managers? Should we feed all of our issue tracker state into an algorithm and get a machine-generated list of assignments for every developer on the team? I may suffer from a lack of vision, but my strong suspicion is that nobody would actually do that - the amount of supplemental information that goes into real priority decisions is substantial. Priorities are often chosen considering things like: Who is expert on this task? Who is going on vacation? Who is overloaded? Who has commitments to other projects? Who are we trying to train to be able to tackle this type of task in the future? Not only are these contextual details not present in an issue tracker, they probably should not be.

If the statistically-impossible but best-theoretically-possible result leaves us wondering how to apply said result, we should find more applicable avenues of research.

4. Small Audiences or Rare Applicability

Consider a recent paper that touched on estimating error bounds in a proof-of-concept co-simulation system. In this case, my concerns are not (purely) whether that's a real problem, nor what to do with the results. My issue here is that the fraction of software engineers concerned with simulation is pretty small in my experience. Further reducing that to people facing the need to combine simulations running in distinct simulation environments starts seeming vanishingly unlikely. Reducing that one more time by finding people that need to be able to bound the magnitude of errors in such co-simulation scenarios, and it's clear that general applicability is unlikely.

In industry there's a term that I'm quite fond of: the "toothbrush test." The most important tools, skills, and ideas are the ones used "as often as a toothbrush." If your chosen problem domain is applicable once a year for a potential audience of a hundred practitioners globally, it should not be surprising when those papers aren't picked up by the industry.

5. Building on Unfounded Assumptions

Several recent papers concerned me by virtue of being fuzzy about what is being used as a ground truth. Considering the Requirements Engineering work again: do we believe that humans are making perfect decisions? If not, why are we training an algorithm to capture that behavior? If we think humans are actually making perfect decisions, we should probably think again.

In a recent Dependency Management paper, SemVer requirements are (unsurprisingly) taken as inputs to the proposed dependency-selection constraints algorithm, alongside call-graph level details. This wholly ignores the fact that SemVer is a lossy human-generated estimate of compatibility for a given change or release, against a generalized abstract notion of "compatibility". This wholly ignores Hyrum's Law, as well as the substantially better detail available from call-graph level detail. It also ignores basic truths about compatibility: compatibility is a property of a relationship, not an entity. Taking a messy, human-generated, lossy estimate of abstract compatibility as a ground truth clearly limits the practical utility of any downstream results. Given this, what makes us believe that the chosen algorithm is giving "better" results in practice?

Or consider a recent paper about evaluating best practices in ML software systems. I generally liked the work, and the inventory of potential practices could be a valuable reference. But when it comes to validating the work, I have concerns. Asking 7 practitioners about whether a given idea is a "best practice" seems iffy, especially when that cohort has a median tenure under 5 years. There are engineers with that much experience who have my complete faith, but most don't. Even ignoring the question of whether the practitioners are qualified to verify the list of practices, we can't really evaluate such things in a vacuum: of course we would prefer to have everything in version control, more robust tests, and better documentation. The question is not whether those practices are a benefit, it's whether in practice we believe it has a positive ROI for the time, resource cost, and effort. If we present a hundred items that someone said was a good idea, it's reasonable to imagine universal support with no constraints or budget.

If software engineering research is about characterizing software engineers and their behavior, we can treat every human decision as correct. If software engineering research is about understanding the multi-version production of multi-version software[5], the notion of ground truth correctness needs to be based

[5]Bonus points for readers that catch the reference to my favorite definition of software engineering, and a mea culpa: I said for a while that this was a Dave Parnas quote but it is in fact properly attributed to Brian Randell.

on what yields the best results. Again we need to be very clear about context and external validity.

6. Academic Toys vs Industry Projects

Lastly, we certainly have to discuss scale. I admit that my years in the industry have messed with my calibration for what counts as “a lot” of engineers, code, data, or compute resources. Still, I suspect that the sizes presented in many software engineering papers are too small for generality or applicability.

Reading through recent JSS papers I saw an analysis of dependency management approaches applied to four low-level packages, none of which had more than 10 direct dependencies, with codebase sizes ranging from 6-116 KLoC. No information was presented about the size of the transitive fanout, if any. Even the lowest-level common packages I’ve worked with are substantially larger and have substantially more complexity than this[6]. Looking at the full dependency tree of a popular OSS desktop system, office, or graphics package would be far more illustrative.

Going back to the Requirements Engineering example mentioned above: methodologically, that paper was based on a single medium/large project at one company. Is there any reason to believe those results have external validity? Was the project successful? Is it representative of the industry at large?

While I have no reason to disbelieve the statistical analyses presented in these papers, the size of datasets, size of programs, size of developer populations, etc., all seem too small. Even if all the other risks listed here are addressed, scale is a substantial concern. And unfortunately, it is a concern that I have few ideas how to manage: industry is notoriously private, for good reason, but without industry-sized problems and populations to study, it’s hard to draw meaningful conclusions.

7. The Meta Question: What Are We Studying?

Lurking beneath several of the above scenarios is an existential question for the research community: is Software Engineering research studying the artifacts and practitioners of software, simply to describe them as-is, or is it contributing to theory building to improve software practices, tools and processes?

If we believe that practice and practitioners are immutable subjects to be approached non-intrusively, that is inherently going to limit applicability and relevance. If we reconceptualize software engineering research as providing theory and evidence for tools, techniques, and practices that lead to better outcomes, that will produce far more applicable results.

[6] See Abseil in C++ or Guava in Java.

8. What Should We Be Researching?

Taking the above concerns into account, the domains I find most interesting are the broad and shallow topics in software engineering. I'd be thrilled to see a general trend toward more overall study and understanding of the broad field, rather than narrow focus on a tightly-focused topic. Broad areas that deserve attention include:

- Productivity
- Testing / defect detection
- Code review
- Prioritization / requirements planning
- Design
- Communication / collaboration
- Version control
- Dependency management
- Release management
- Reliability

These are topic areas that matter to most engineers, most of the time. Many of these topics pass the toothbrush test. There are endless unanswered questions in these domains, as well a staggering amount of misunderstanding and misinformation among both practitioners and researchers. A substantial shift in priority toward these topics would be wonderful, and (I suspect) would majorly improve the applicability of research in this domain.

I cannot claim to have a complete listing of relevant research questions in those topics. However, we certainly can zoom in from those broad topics to some research questions that I would be *thrilled* to have answers to:

- What's the project size/scale/lifespan payoff rate for automated unit testing? That is, how would we describe the tipping point where (on average) it's more efficient to write tests than not? Most engineers I have asked estimate that it's on the order of 1 engineer x week, but a convincing study would be useful. While automated testing and continuous integration have had substantial uptake in the industry, it is by no means rare to still encounter organizations with little to no testing or test automation. Providing additional citations and evidence on the efficacy of this approach would be hugely impactful.

- What are the pros and cons of adopting code review policies? There have been horror stories of teams with toxic (or entirely perfunctory) code review cultures, but there are also organizations where it's a substantial, and important, piece of the software workflow.

- What fraction of SemVer constraints are bogus? As Russ Cox points out, most specified SemVer dependencies are decided based on whatever the developer happened to have installed at the time. That isn't actually evidence that an earlier version wouldn't work. Nor is the removal of an unused API evidence that a major version is incompatible[7]. The only truth in these "compatibility" discussions is to build the code and run the tests[8].

- What are the generally accepted best practices (version control, CI, code review, etc)? How consistent is adoption of those practices? Do adoption rates or practices differ by language or industry? How are those changing over time?

- What measures of code quality match human intuition? How much can that be automated, if at all? (And if we automate that, do we run afoul of Goodhart's Law?)

- What proxies do we have for engineering productivity? In which domains do different proxies apply?

That final question is perhaps the most important. Although we likely take it too far and are too reductive in our approach, industry practice is sensibly focused on impact and good return on investment. Anything that academic software engineering research can do that helps improve outcomes, reduce costs, improve efficiency, or measure productivity has a decent chance of seeing some uptake. If we want to make research venues like JSS generally applicable to industry practitioners, the density of papers that help with common real world tasks needs to increase. As it stands, the ROI for *reading* through these papers is generally lacking. If I (and *all* the academia-adjacent practitioners I count as friends and colleagues) can't justify spending the time on that reading, that tells us everything we need to know about the current gap between academia and industry.

[7]Removing an unused API is a perfectly compatible change.

[8]My intuition is that a majority of the time that we are dealing with "dependency hell" is due to irrelevant SemVer constraints. The industry as a whole will be grateful if someone can prove that the current path is silly.