The Case for Context-Driven Software Engineering Research

Generalizability Is Overrated

Lionel Briand Domenico Bianculli Shiva Nejati
Fabrizio Pastore Mehrdad Sabetzadeh

SnT Centre - University of Luxembourg
{briand,bianculli,nejati,pastore,sabetzadeh@svv.lu}

This article follows up on Lionel Briand's 2012 Sounding Board article on the significant disconnect between research and industrial needs in software engineering [1]. Here, we argue that for software engineering research to increase its impact and steer our community toward a more successful future, it must change. Specifically, we see the need to foster *context-driven research*. By that, we mean research focused on problems defined in collaboration with industrial partners and driven by concrete needs in specific domains and development projects. By analyzing publications from the top software engineering research venues, anyone could easily conclude that only a small proportion of the papers stem from such research.

Context-driven research doesn't try to frame a general problem and devise universal solutions. Rather, it makes clear working assumptions, given a precise context, and relies on trade-offs that make sense in that context to achieve practicality and scalability. This research paradigm applies to any topic in software engineering and isn't meant to highlight any particular research area. Because context-driven research doesn't produce results that generalize easily to any arbitrary software development environment, the following questions arise: Does it have value? How so? Is it (engineering) science? This is what we discuss in the rest of this article.

The Importance of Context

The main motivation for the proposed paradigm shift is that software engineering solutions' applicability and scalability depend largely on contextual factors, whether human (such as engineers' background), organizational (such as cost and time constraints), or domain-related (such as the level of criticality and compliance with standards). For example, whether a verification technique's cost is justified will depend on the criticality of the software being assessed and the standards it must comply with.

For many safety-critical systems, standards require traceability between requirements and system test cases. This naturally leads to the development of techniques supporting the definition of requirements enabling the automated or semiautomated derivation of test scenarios.

Testing and verification techniques typically take a specific set of inputs—such as a model or source code—and make assumptions about those inputs' form and content—such as a model's level of precision or the constructs in source code. Whether such assumptions hold typically depends on the type of system, the development process, the software engineers' background, and the cost and time constraints.

These techniques' outputs usually are assumed to be useful to support a given development task; whether this is true also depends largely on contextual factors. For example, many software engineering research papers on cyber-physical systems (CPSs) assume (often implicitly) that the data sent to actuators or received from sensors is Boolean (or enumerations), or they exclude the notion of time. Pure Boolean abstractions are valid for a certain class of CPSs and for some specific analytical purposes but are simplistic in many other cases. This leads to overly coarse models that enable only shallow analysis of CPSs. Such unwarranted simplifications and generalizations are often due to community biases that assume any computation is rooted in logic. Our experience is that context-driven research can correct such biases and bring realism to research.

A common misperception is that long-term, high-risk research (sometimes called basic research) must be free of practical considerations. A widespread and sometimes implicit argument is that we, as researchers, need to make simplifying assumptions to work out the fundamentals, which can then be tailored to more realistic contexts. However, solutions developed that way are rarely adaptable to any real context. Usually, it's infeasible to modify such an approach or technology to relax assumptions about its inputs, scale, people's background and capabilities, or any other relevant aspect.

No fundamental contradiction exists between doing basic research, having realistic assumptions, and accounting for context in defining our targeted problems. Doing basic research in no way prevents us from grounding our work in reality. For example, observation of current engineering practices in the CPS domain reveals that most analyses rely on continuous-time-signal representations of CPS behaviors. This triggers the need for challenging basic research to devise proper modeling abstractions and analysis techniques for CPSs that account for their continuous and physics-based aspects.

Nevertheless, someone might argue that context-driven research leads to results that don't grasp the big picture and to solutions of limited relevance. But what's the alternative, which has driven much of software engineering research to date? Research in a vacuum? Research driven by purely academic considerations? Based on many years' experience in research and engineering practice and dozens of collaborative industrial projects, our position is that such research has a limited impact on the industry and society it's supposed to eventually serve.

The main reason for this limited impact is that research in a vacuum relies on assumptions that are unlikely to ever match any real context and therefore lead to impact. In the best cases, such research will address the exception rather than the most common situations. For example, regarding quality assurance for natural-language requirements, numerous assumptions can directly affect a technique's applicability, such as the availability of a domain glossary, the use of strict templates, or a certain amount of background knowledge. More specifically, and as an instance, if a technique relies on the availability of a complete glossary, it's unlikely to be usable in most contexts in which such a prerequisite isn't met.

Furthermore, although contextual factors strongly drive the applicability of techniques and methodologies, software development organizations tend to be representative of certain application domains and develop types of systems that are also common in other organizations. For example, the development of controllers in embedded systems tends to present similar challenges and relies on similar technologies across companies and domains, such as modeling in Matlab or Simulink and generating code. Also, many information systems implement business process models and rely on technologies driven by the Business Process Model and Notation, whose cross-domain adoption has risen steeply over the past decade. This implies that context-driven research's results tend to be relevant outside the immediate context in which they were obtained, usually in some type of industry or domain.

An additional benefit of context-driven research is that immersion in a domain and context helps identify problems that academic research has overlooked but that are nevertheless important. For example, while working on the specification and testing of systems with strong legal and compliance requirements (for instance, a tax or social-security system), we observed a serious communication gap between software engineers and legal experts. The former usually have a superficial understanding of the law; the latter can't analyze and understand source code and complex formal notations. So, it's important in such contexts to devise a domain-specific modeling methodology that both types of stakeholders can understand.

Because our premise is that contextual factors vary widely across domains and industries and therefore no universal solution exists for most software engineering problems, we suggest that context-driven, bottom-up research should play a more prominent role. In short, this approach entails that we solve problems in context, identify commonalities and differences across contexts, adapt solutions to different contexts, and generalize over time by building a body of knowledge from concrete experience.

What Must Happen

To achieve this vision, several things must change in our research community. First, top journals and conferences must acknowledge that context-driven research is needed and challenging, and must review it accordingly. Guidelines could be developed to help educate reviewers and the research community. For example, some of these guidelines could focus on how to request evidence and arguments about a proposed approach's correctness and completeness and when to request additional case studies.

In reviewing context-driven research, we must distance ourselves from the research tradition in computer science that emphasizes proofs of correctness and completeness instead of practicality and scalability, and that tends to believe in universal concepts and solutions. Mathematically demonstrating the correctness and completeness of practical, scalable solutions is rarely possible, similarly to proving the correctness of an industrial-strength software system. A clear rationale and clear arguments, thorough testing, and, when possible, the public availability of tools are usually the way to go.

Furthermore, funding agencies must help promote and reward collaborative research with industry, thus enabling context-driven research. It's unrealistic to believe that such research will expand if financing it remains difficult. Finally, hiring and promotion committees must move away from counting papers and citations and instead emphasize and reward evidence of impact on engineering practice.

Devising software engineering solutions in a vacuum, in the comfortable setting of academic offices, is unlikely to yield the results the software industry needs. Our research must operate in clearly defined contexts, enabling us to identify realistic working assumptions and identify important, well-defined problems, as well as create opportunities for realistic evaluations. The fact that solutions and results don't generalize to all contexts shouldn't be of concern. We must accept that we're an engineering field impacted by human, domain, and organizational factors and that universal solutions are virtually nonexistent.

Eventually, when developing and adapting solutions in various contexts, based on empirical results in a variety of settings, we'll be able to derive a body of knowledge that helps practitioners determine what to use in their context. Until we promote and publish such context-driven research much more than we do now, the gap between academic research and industry needs will remain, and the former's impact will remain limited. Our proposed paradigm shift—inspired by the engineering roots of our discipline and breaking away from computer science research tradition—is a significant challenge, like all endeavors requiring a change in mindset. To improve the situation and ensure a bright future for our research, we all have a role to play, including academic researchers and leaders, academic institutions and departments, and funding agencies.

References

[1] Lionel C. Briand. Embracing the engineering side of software engineering. *IEEE Software*, 29(4):96, 2012.