Position Statement: Ceaselessly-Analyzing Development Environments, One Direction For the Next 40 Years of Software Engineering

K. Rustan M. Leino
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
leino@microsoft.com

Software engineering is both difficult and expensive. This is as true today as it was 40 years ago. The end products of software engineering, software artifacts consisting of streams of instructions, are intended to solve some problem, live up to some requirements, implement some design. To improve software engineering, we must continue to improve the process that eventually outputs these instructions. We want the instructions correctly to address the problem, and we want the ability flexibly and precisely to change the instructions when the problem, requirements, and design change.

Most interesting problems are far too complicated to let us produce instructions directly from high-level descriptions of the problems and their requirements. Instead, we use a number of design layers, each getting closer to the eventual instruction stream. It is well known that the cost of correcting an error in a software artifact is lower the higher up in these layers that we detect the error and the sooner in the design process that we do so.

In the last 40 years, we have made progress in the lowest layers of this design process, developing programming languages and methods that avoid certain kinds of errors. For example, structured programming helps us reason about control flow, type systems help us detect inconsistent usage of data, and garbage collection simplifies our task of defining interfaces between modules. A number of analysis tools that check the resulting programs for errors, both shallow and deep, have proved useful. Quality is assured and assessed mostly through testing, but the test tools have become more advanced, not just in orchestrating the tests, but also in automatically generating tests from specifications and code.

We have also learned how to apply language technology at a layer above traditional programming languages. For example, various software specification and modeling languages can express a design at a higher level, and the last decade or two have shown progress in tools that analyze such models to prove properties and find errors, in machine-assisted refinement frameworks that allow the specifications to be turned into compilable code, and for some domains even in automatic compilation techniques for specifications.

The availability of tool support has been a major ally in our wrestle with the increasing complexities of software engineering in the last 40 years. Tools are still becoming better and more plentiful, and I predict we will continue to rely more on various kinds of tool support in the future. The tools will become more effective at finding specific kinds of errors and will become more generally applicable. The additional tool support will also let us advance the design of meaningful languages for use in higher layers of the design process: specifications, models, and requirements.

In the future, we will have far more processors and processor cycles than we do today, so I envisage that we will also create development environments that run the tools continuously in the background, like a horde of detail-oriented tireless colleagues who, peeking over our shoulder, incessantly study our models and programs. By ceaselessly analyzing our programs, these development environments will be able to find errors in models, automatically generate and run test cases, symbolically analyze program texts against specifications, and attempt compilation strategies that give us code for parts of the system that have been specified but not yet implemented by a programmer. As more parts of this process become automated, we will better be able to produce correctly functioning software, even when problems, requirements, and designs change.