My predictions for the state of the art in programming technology in the year 2030

CS208 Project 2

Philip Guo (pg@cs.stanford.edu)
June 2, 2010

1 Purpose

In this paper, I offer my predictions for what the state-of-the-art work environment for professional programmers might look like in the year 2030 (20 years in the future). Out of all possible domains in which computing will make revolutionary advances in the next 20 years, I’ve chosen to focus on advances in technologies surrounding the act of programming, since programming underlies all innovations in computing. The one action that innovators in computing fields as diverse as artificial intelligence, distributed systems, graphics, user interfaces, and web search must all perform is that they must all write computer programs to turn their ideas into reality. Thus, technologies that help people write programs that are more reliable, maintainable, and malleable will help accelerate advances across all areas of computing.

2 Modern-day programming technology

Before speculating about the future, I will first describe the current state-of-the-art work environment for professional programmers in the year 2010. A typical modern-day programmer works on a 2 GHz desktop computer with a 24” LCD flat screen monitor. He\textsuperscript{1} spends most of his work time interfacing with an IDE (Integrated Development Environment) on his computer.

\textsuperscript{1}For simplicity, I will use the masculine pronoun throughout this paper.
Within the IDE, he can browse through the entire project tree of source code, navigate via auto-generated ‘hyperlinks’ to jump across files (e.g., by clicking on a function call site, he can navigate to the function’s definition), write and compile code, run automated tests, and debug via an interactive source-level debugger. He also spends a fair amount of time in his web browser, searching both the public web and the company intranet for documentation, tutorials, technical discussion forums, and code examples (in addition to indulging in non-work-related distractions). He manages bugs and feature requests using a bug tracking system and organizes versions of his code using a revision control system, both of which he can access from within his IDE. Lastly, he constantly uses email, instant messenger, and video chat to coordinate with his project teammates (who are sometimes on the other side of the world).

The modern-day programmer has far more productivity technology at his fingertips than his counterpart from 1990 (20 years ago). A programmer in 1990 was probably writing code in a simple text editor on a 30 MHz machine with a 12” CRT monitor. His iteration and debugging cycle was often quite slow, since compilation and tests took much longer to run due to slower hardware. Compiling code after a small change often took several minutes, a stark contrast to the nearly-instant compilation times that programmers today take for granted\(^2\). Also, the web didn’t exist back in 1990, so programmers weren’t able to search online for documentation, tutorials, or example code. Instead, they had to rely on code comments and design documents (which are tedious to update and thus prone to getting outdated). Lastly, instant messenger and video chat didn’t exist, and corporate email was in its infancy, so it was much harder for programming teams to coordinate.

Although technological advances in the past 20 years have undoubtedly made programmers more productive, they have not done much to address the core problems that make programming difficult. In particular, it is . . .

- still difficult to create comprehensive test suites
- still difficult to find bugs and security vulnerabilities in code
- still nearly impossible to prove guarantees about properties of code
- still hard to find the proper documentation, example code snippets, or person to consult when trying to understand a software component

\(^2\)A modern-day IDE like Eclipse can incrementally compile code while the programmer is typing, thereby providing instant feedback for syntax or type errors.
Today, all of these tasks must be done manually by humans, which is slow, error-prone, and costly\textsuperscript{3}. For instance, programmers often look for security vulnerabilities by meticulously reading through reams of printed-out source code. However, I am optimistic that in 20 years, there will be tools available to automate parts of these tasks, so that programmers can focus more on the creative aspects of their jobs and less on bookkeeping and logistics. There has been a great deal of academic research in the past decade with these goals in mind, so hopefully those technologies can transfer into the mainstream within the next two decades.

\section{Programming technology in the year 2030}

I predict that drastic improvements in automatic test case generation, software bug finding, code verification, and information retrieval tools will make it far easier for a programmer in the year 2030 to quickly write and revise software that is reliable, maintainable, and malleable. Informed by a literature review\textsuperscript{4} of academic research from the past decade, I now present my vision of the programmer’s work environment in the year 2030.

The programmer of the future will have access to several orders of magnitude more computational power than available today. While CPUs might not actually get much faster due to physical limitations, they will be so cheap that a programmer’s desktop computer will have hundreds of CPUs. His computer will also be connected via high-speed network to all of his colleagues’ desktop computers (each with hundreds of CPUs) as well as a local data center with a computer cluster containing tens of thousands of CPUs.

However, the basic actions of reading and writing source code, browsing documentation, and electronically communicating with team members aren’t too computationally intensive, so what can be done with all of that latent computational power? It seems like an enormous waste to have most CPUs sitting idle throughout a programmer’s work day.

\textsuperscript{3}Programmer time is far more expensive than machine time. Also, computer hardware will get drastically cheaper over time, but human labor costs will not.

\textsuperscript{4}Disclaimer: This brief literature review is far from comprehensive; it’s strongly biased towards research done by my friends and colleagues. Fortunately, the bibliographies of these papers provide a more comprehensive survey of the field.
3.1 Automatic test case generation

With so much free computational power, code compilation will be instantaneous, and even complex test cases will run fairly quickly. Thus, the IDE will constantly be running as many relevant tests as possible all the time while the programmer is working, which will provide nearly-instant feedback when semantic bugs arise. Saff and Ernst empirically demonstrated that when unit tests are run automatically and continuously, programmers not only save the context-switching time of manually running tests, but more importantly, can identify and fix bugs more effectively since bugs are exposed as soon as a line of code is edited (rather than tens of minutes or even hours after the code that triggered the bug got written) [15].

The act of writing unit tests is often tedious and boring, so why not have the computer automatically generate unit tests directly from the code itself, while the programmer is writing the code? Several research prototypes in the past decade have demonstrated the feasibility of this proposal: Pacheco et al. use heuristic random testing to generate sequences of method calls for object-oriented unit tests [13]. Cadar et al. use a fine-grained constraint solver to generate detailed tests for low-level systems code [5]. These techniques are fully-automatic but extremely CPU-intensive; fortunately, since their algorithms can easily be parallelized, they can take advantage of the hundreds of CPUs on a programmer’s desktop machine and the tens of thousands more accessible over the network (on other programmers’ idle machines and in the data center). Sure, many of these automatically-generated test cases won’t be too valuable, but since they are created without any human labor, they are essentially free5.

Of course, some tests must still be written by a human, since they evaluate higher-level semantic properties that might be difficult to automatically infer from the code itself. However, the computer can still assist programmers in writing and managing such tests. For instance, if these tests (e.g., end-to-end system tests) take too long to run, the computer can automatically split them into smaller sub-tests, each of which can be small enough to run instantaneously while the programmer is editing his code (see research in test factoring by Saff et al. for an early example of this technique [14]).

5modulo energy costs, which are still much cheaper than programmer or tester salaries
3.2 Automatic bug finding

Modern compilers perform some basic static code analysis to catch superficial bugs like when a program has a type error, attempts to read an uninitialized value, or shadows a variable with the same name from an outer scope. Compilers of the future will be able to perform far more sophisticated static analyses that can find deeper bugs, thanks again to enormous available computational power. These futuristic compilers will be able to instantly compile code and report bugs like:

- This pointer dereference is likely to lead to a crash on this code path when these particular boolean conditions are met [7]
- There should probably be a call to foo() here, since there was a call to foo() in 100 other pieces of code that resemble this one [8]
- This is likely the source of a security vulnerability [16]
- There is likely an infinite loop under this precise condition [6]

Finding these sorts of bugs can take exponential time, so in order to terminate in a reasonable amount of time, modern static analysis tools must make approximations and use timeouts. These imprecisions lead to false positives, thereby diminishing the tools' effectiveness. Even with these imprecisions, their algorithms usually still take hours or days to terminate on today's hardware. However, the good news is that the more computational power is available, the fewer approximations are necessary, thus leading to greater precision and fewer false positives. In the future, these analyses will run on thousands of machines in parallel, so that they can give results within a few seconds (as fast as modern compilers can compile code and provide simple warnings). Ranking heuristics and adaptations to user feedback can mitigate the detrimental effects of the remaining false positives [11].

A complementary approach to using static analysis to find bugs is to use dynamic analysis — executing the code repeatedly under varying conditions in order to uncover bugs. The key advantage of dynamic analysis is that there are no false positives, since the actual code is being executed (rather than a statically-evaluated approximation of the code). Unlike static analysis, which can only provide likely warnings, a dynamic analysis can generate concrete inputs that expose a bug (often at the expense of longer running
times, though). Here are some examples of dynamic analyses that could be integrated into the daily workflow of the future programmer:

- Automatically generating SQL injection and cross-site scripting attacks for web applications [10]
- Automatically perturbing the OS scheduler to uncover concurrency-related ‘Heisenbugs’, which are notoriously difficult to reproduce in unit tests and nearly impossible to find using static analysis [12]
- Using a directed fuzzing technique to automatically create malformed input files that trigger crashing bugs and security vulnerabilities in programs with complex input data parsing code [9]

It might take hours or even days of 10,000 machines crunching at full capacity before useful bugs are found, but once again, machine time is far less costly than human time. These longer-running dynamic analyses can be run continuously (in parallel) in the background while programmers are working (or sleeping). Even a well-paid QA specialist might not want to suffer through the mind-numbing tedium of trying to reproduce a concurrency-related ‘Heisenbug’, but the machines will never complain.

### 3.3 Code verification

The ‘Holy Grail’ of program analysis is the ability to verify (indisputably prove) certain properties about code, such as “there will never be an unsafe pointer dereference that crosses a user-kernel permission boundary” [4] or “this section of code will always terminate” [6]. It’s beneficial to automatically find bugs like “here is an unsafe pointer dereference where a user pointer is directly dereferenced in kernel code”, but it’s far more useful to prove that no such bugs exist anywhere in your code base. The only sure-fire way to tell that a piece of software is completely reliable (with respect to a certain property) is for it to pass formal verification. For realistically-sized pieces of software, no amount of testing or code inspection can accomplish this goal.

However, as expected, verification is a far tougher problem than bug finding. But again, I am optimistic that advancements in theoretical techniques combined with massive increases in computational power over the next 20 years will make certain kinds of code verification commonplace. Today, we run nightly system-wide regression tests, but in 20 years, we will be able to
run nightly verification proofs. I imagine that every evening while programmers are at home sleeping, all of their desktop machines and data centers will be connected together into a cluster of hundreds of thousands of CPUs cranking simultaneously on verifying some property of the latest version of the code base. Then when programmers return to work the following morning, they receive an email report showing which properties still remain properly verified and which ones were broken due to the prior day’s code edits.

3.4 Information retrieval

So far, I’ve only discussed what happens after a programmer writes code: test cases are automatically generated, sophisticated static and dynamic analyses are run to find bugs, and perhaps some verification proofs are performed. But programmers spend a great deal of time trying to simply figure out what code they should even be writing. Programmers must retrieve and comprehend information about the given software system and its multitude of components before they are able to make effective contributions to it.

Today, this information retrieval task is a tedious manual process. A programmer who is new to a software project starts by reading its code, then tracing through its dependencies and reading more code, and then maybe emailing some people to ask for help or searching online for tutorials, discussion forums, or other documentation. I predict that in 20 years, computers will fluently assist in these programming-related information retrieval tasks.

First, just like how Google automatically crawls and indexes web content, automated agents will monitor and index how each programmer interacts with his code, documentation, and other work artifacts, and then make that information available to all other programmers. When a new programmer is browsing through a particular piece of code, his computer can make informed suggestions for who are the best people to contact if he has a question about that code. Begel et al. showed that the seemingly simple question of “Who is the best person to ask if I have a question about this code” can actually be tough to answer in a corporate software development team, oftentimes leading to wasted employee time and corresponding productivity drops [1]. In that same paper, they present a prototype system called Codebook that provides some computer assistance for this human-centric search procedure.

When it’s not feasible to ask a person for help, online textual information (either on the public web or within the company intranet) can help programmers understand what code they need to read and write. In particular,
when a programmer is learning a new language, API, or library, he often performs ‘copy-and-paste’ programming by reading example code snippets, copying and pasting them into his IDE, and tweaking the examples until they work properly for his particular use case [3]. In the future, there will be a far tighter integration of code examples and documentation into the IDE, so that it will be easy and intuitive for programmers to locate, copy, and integrate example code into their projects (Brandt et al. built a prototype of such a system at Adobe Labs [2]). These systems will be able to extract code snippets not only from full-fledged ‘official’ examples on the web, but also from the code bases of project colleagues and open-source code repositories.

4 Conclusions

There have been many advances in programming technology in the past 20 years, but we are still performing some of the hardest parts of programming without much computer assistance. It’s still difficult to generate good test cases, to find subtle bugs, to convince ourselves that a piece of code works properly, and to even figure out what code needs to be written in the first place. In the next 20 years, advancements in the research I surveyed throughout this paper, combined with massive increases in computational power, will enable computers to more effectively assist programmers in writing high-quality code in less time.

References


