

Software Engineering as Live Performance

Richard P. Gabriel

IBM Research

Hawthorne, New York USA

rpg@ $\left\{ \begin{array}{l} \text{us.ibm.com} \\ \text{dreamsongs.com} \end{array} \right.$

ABSTRACT

Future software systems will be vast and impossible to rebuild. The tools engineers need to get them and keep them running need to take advantage of the best we have in static and dynamic languages—to begin with. Long-running systems must be repairable and extendable while they run. We can leverage this longevity by designing our languages and systems to learn about and create models for themselves, to hypothesize improvements on themselves, discover and propose new capabilities, and to conscientiously assist in their own upkeep and continual redesign. All while the system never stops. Our lives will depend on it.

Categories and Subject Descriptors

D.2 [Software Engineering]; D.3 [Programming Languages]

General Terms

Design, Reliability

Keywords

Design, programming, self-healing, self-sustaining, self-designing

Here are some of the ~~unpleasant~~ exciting things I believe will be true about our future systems—that is, the ultra-large scale ones that will grow to a size that prevents them from being stopped and reinstalled, that need to be running continuously or bad things will happen, and that must be repaired and extended only while they are executing. They will not be fully engineered; integration of the existing systems that make them up will display significant emergent behavior that's hard to predict; there will be no way to fully

test or beta test what is deployed; the systems will be too big to fail; we won't know how to design them, or even be sure what design is in this reality; many of these systems involve civil and above all human safety; adversaries will continually attack such systems, be those adversaries people on the attack, machines on the attack, or errors, faults, and failures on the attack; requirements will not be consistent, and sometimes requirements will be way out of date and stakeholders long departed if not dead; there will be a larger gap between such ultra-large-scale systems and enterprise-scale systems than between enterprise-scale systems and individual applications; chaos and inconsistencies will abound; failures will be constantly occurring; people will be part of the system, and inextricably so; and finally, such systems will be beyond human comprehension [1].

The questions are: how do we get one of these systems going and how do we change it as change is needed? As software engineers our hands are tied because the tools we use are typically defined by computer scientists who have an old-fashioned concept about what software is and how it's created. Their idea in strawman form is that someone prepares a text which is a static representation of executing software; that text is checked to make sure it is written correctly (that is, it is in the form of what programming language theoreticians believe is proper for software), then that text is transformed into executing software, and finally the running software is judged a success, a failure, or something in between; later, the process roughly starts over. This is nothing like how ultra-large-scale systems will need to be treated. Naturally there are lots of tools that address the real construction of running software systems, but it's still true that most "real" programming languages center around what the compiler "thinks" about source code presented to it, that performance concerns dictate minimizing runtime representations and overhead, and that all the other tools need to work around these realities.

Each ultra-large-scale system will be installed at most once. And once installed, it really cannot be stopped—too much depends on it staying up. The source text we're used to dealing with is like a genetic code in that it describes what will be created in the form of a running system—it's the genotype to the executing code's phenotype—but this doesn't help us repair or extend something that's already alive (doesn't help

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FOSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

Copyright 2010 ACM 978-1-4503-0427-6/10/11...\$10.00.

us much). When I get a cold, my mother doesn't gestate a variant of me that doesn't have a cold and somehow that near clone replaces me in the real world. I take some medicine to alleviate the symptoms and then my body fights off the cold and heals itself. How would this translate to what a software engineer would do?

Looking at some answers to this question will lead us to a proposal about future software engineering research.

Suppose it's the future; suppose software is still represented as source text, and a component of a running system is found to have a problem. The first thing might be to figure out what conditions cause the problem or what observations signal the problem is about to happen or already has happened. Then the engineer could add some code to the running system—a patch—that detects that bad situation and repairs it on the fly or otherwise avoids it. One could think of this as a wrapper around the faulty component, but other techniques could work. If the problem causes a component to fail, perhaps that component could be rebooted; if a combination of inputs and environmental conditions indicate a failure is about to happen, perhaps that combination can be avoided, or different actions taken when they are detected, or the return values of compromised components could be adjusted. Thereby the symptoms of the problem could be temporarily alleviated until a true fix can be found or constructed.

If the engineer can locate the problem and propose new source text that would fix it, the next step would be to determine whether the repair will work. Remember: the system cannot be stopped or paused significantly; perhaps lives depend on the system operating at least as well as it is now. The first line of attack should probably be offline.

Maybe one thing to try would be to create a simulation of the running system with plausible inputs and environmental conditions derived from statistical models of the running system *in situ*, with the proposed new component plugged into the simulation. Because the system has been running for a long time, there should be good models for most or all of the components in the system. Perhaps, for example, the behavior of a component can be modeled by a particular satisfiability problem, and a sat-solver can be used in place of the component. Using these models the engineer can perform a range of testing that maybe is not so easy to do otherwise because the component is not so easily isolated. One way that can happen is that the component could be *ephemeral* in the sense that no abstraction exists for it in the original source code and therefore also not in the executing code aside from patterns that can be recognized. Perhaps there is no way to encode the ephemeral component in the original programming language. Nevertheless, abstraction in the sense scientists use the word can be performed—recognizing a pattern, identifying its common and variable parts, and presenting the recognized pattern in a canonical form.

Here's what I mean by this. Suppose a piece of software is written using the observer pattern from the Design Patterns

book [2]. If done in a usual way in a usual programming language (today), this will result in a customized instance or instances of a configuration of objects interacting in patterned ways. It is possible, though, for a sophisticated pattern matcher to recognize these patterns and present imagined source code with the observer pattern explicit. This has been called “registration” by some [3].

Our engineer, then, would be able to program in these perceived patterns, and the results can be installed—via a detailed surgery—into the simulated system (and later into the running system).

These experiences will give the engineer some confidence that the new component will work ok. But perhaps not enough. The system cannot be stopped, so the engineer must gingerly determine whether the new code can be substituted, so the next step might be for the changed code to be introduced into the system (which we can assume is currently running ok—the temporary patch still holding), but in a provisional way, with the problem component (assuming it's a component) in place to take the bulk of the load. All the real inputs could be sent through the existing component (and its patches) while the new component is turned on only at specific times and places so that the engineer can watch what happens. Perhaps the inputs are given both to the old and new component and the (differing or same) results can be observed, maybe with the original component still supplying the real results.

As confidence grows, maybe the engineer turns on a mode in which the results of the original and revised components are fed in pairs to the rest of the running system and a display of differential effects is shown so the engineer can evaluate the new component.

What this is like is being a guitarist who is joining a blues band noodling around with a song with her amplifier down low (or being played only through headphones), and as the song is understood (blues players don't typically use sheet music), that amplifier is turned up and the audience can start hearing the new player's stuff along with the others'.

As the engineer works, the source code—real or imagined—is displayed showing details of historical values for inputs, variables (internal or global), fields, parts of data structures, etc, as well as recent or current output values as the system is running behind the scenes. Proposed changes can be examined as if they had been made (provisionally) in the running system as described. This way the effects of design and implementation decisions can be seen in the running system but without jeopardizing its current satisfactory execution.

As part of the process of (possibly accidentally) designing and implementing the system, it might be sensible to retain all the versions of all components that ever existed—in the running system but possibly not being exercised. These earlier versions were once considered state of the art, and so there might be reasons to want to fall back on one of them. Or perhaps on several of them. Or a combination of them depending on the context including inputs and other observ-

able information. Perhaps one is faster on a certain family of inputs or under certain conditions; perhaps one is really needed only in a particular situation that occurs infrequently; perhaps one is ultra-reliable under all conditions, but runs slowly; perhaps one is guaranteed correct because it transforms the problem solved by the component into a problem that is explicitly solved using a general technique (like the sat-solver mentioned earlier); perhaps one runs very quickly but doesn't produce much accuracy (a numeric component). It might be sensible then to have a decision tree or neural net or digitally evolved program look at the invocation situation and select the right combination of versions of the component to exercise. Moreover, having a set of versions of the same component around, each with a different set of properties, can shed light on the purpose and history of the component and thereby of the whole system itself.

As the system runs, a large set of test cases can be gathered—both positive and negative tests—which can be used to direct searches for the best combinations and selections of the existing components. This is a form of directed evolution with the accumulated test cases acting as a fitness function. And these tests can also be used to find other components (or ephemeral components) that accomplish the same purpose.

When the possibility of combining and selecting behavior, possibly based on machine learning, is available, it becomes possible to think about refactoring components and recombining the factors differently to obtain candidate replacement components that might have some advantageous characteristics.



What's being described is a programming system—particularly the “runtime” portions—that is more suitable for *in-situ* observation, diagnosis, and modification than most of the ones software engineers currently have access to. In particular, I'm talking about systems that are more aware of how they are put together, how they are operating, and how to modify themselves. Presently, the primary research efforts in programming languages focus on how to produce the smallest, fastest set of executable bits that will get a well-envisioned piece of functionality running. There is little interest in self-correction, self-repair, and self-awareness. Not none; just little.

But before you label me a pure dynamicist, consider that the benefits of static typing and being able to reason about large portions of a system at once should not be given up on. In the always-on vision of systems in the future, I see the source text there too, with all the benefits of static descriptions sitting side by side the benefits of dynamic observation. Why not when you visit Firenze have a guide book with you as you visit its glories? And why not when you observe the

actual runtime type of an object also observe its declared / expected type—and be able to act on that information as well?

In the future imagined in the ULS report [1], ultra-large-scale systems will be created at least in part by putting systems together, such systems perhaps not designed with the others in mind. When new systems are added to an existing system, it might turn out that there is a capability that the newly added system has that is similar to a capability the old system has and uses, and it would be useful if the existing system could notice that and either recommend using the better version or even wire that up itself. What I imagine is that the provisional try-out mechanisms just described can be exploited by the executing system itself to learn how to use the new version and try using it out without any bad effects until the system has confidence the new version is working well or has been adapted to do so.

This means that the running system should be at least a bit self-organizing. Further, the running system should exhibit *quenched disorder*, which means that almost all executions run through the usual execution paths, but every now and then a random execution tries to find (in a provisional way) alternative execution paths, based on similarity of functionality as determined by what the system is learning about itself while it's running—so, the system should be continually gathering and refining ideas about the “function” of each component, what sorts of inputs components use, alternate ways to accomplish its various purposes, and proposing if not implementing improvements. These ideas can be couched in terms of the test suites for different components (or larger structures) that are being generated as the system runs.

In a similar fashion, the system should learn which inputs and components are involved in problematic executions, and learn how to alert someone (or something) about the problems, log them, and maybe learn to route around the problem or adapt to provide proper behavior. As noted, there should be lots of test cases gathered over, let's say, a decade of executions, and maybe different models will have been learned that can mimic components in the system.

Most of these ideas are about making existing functionality work better and more reliably, but there is another avenue: using the system to explore how it could extend itself. Recent work in evolutionary algorithms has shown that in at least some cases it is possible to find effective and advantageous algorithms based on searching for *novelty* rather than *fitness*. Such systems use a variation on genetic algorithms but replace the usual fitness function that tells the genetic algorithm how close it is to finding a solution with a novelty function that tells how far a solution is from existing ones. The surprising result [4] is that the approach of using a novelty function within a constrained problem space but with behaviors that are relevant to that space can find solutions when the fitness-function-based approach cannot, and with fewer generations when both can find solutions. The basic reason is that a novelty seeker is not deceived by local optima—it

is generally looking to do something different rather than get close to something. In a deceptive environment—which is filled with local optima—finding novel behaviors is more likely to find a way to the real goal than trying to tune an apparently nearby solution.

In a ULS system, it might make sense for the system to look for novel behaviors it can perform, and to propose them to engineers and stakeholders. For such an approach to work, there might need to be a way to prune novel behaviors for utility.

What this all means is that an executing ultra-large-scale system should not be a machine off in the wilderness just chugging away, but it should be a self-aware, self-organizing, self-healing, nearly conscious system that is preparing itself for adaptation to the future.



But hold on you're saying. This isn't like what programming is like today. That's right. Programming is viewed as a *de novo* exercise, whereas here it's a modification / modulation process. For a real engineer in the future, the world will consist of modifying a system that cannot stop, whose proper functioning is required all the time.

This is because the agenda of programming language design and environmental support precedes that of software engineering—"precedes" as in has precedence. This has the effect of imposing the theoretical ideas of programming on the practical world, and for ultra-large-scale systems, engineering will be very different from what is imagined in this historical approach: programming language research and thinking have been predicated on the concepts of correctness, reasoning, proving correctness, efficiency of execution, and preventing errors. In real ultra-large-scale systems, all of those things would be nice, and can be achieved to some

degrees in some (relatively small (but growing)) circumstances, but it's unrealistic to expect only perfect software to be deployed. Moreover, reasoning—which is important both in the old-fashioned way of thinking about programming and in the examples I used above—in the new world of software systems involves not only mathematical-style reasoning (deduction, mostly) but also induction (scientific validation) and abduction (hypothesis formation). This involves sensors, actuators, transparency, record-keeping, automatic hypothesis formation, learning, and the like. A software engineer needs to work with running machinery, and the luxury of a fresh start is simply not available.

What I propose for future software engineering research and work is to imagine software engineering in a future filled with ultra-large-scale systems, to imagine what sorts of languages, tools, and capabilities would be wonderful to have—regardless of whether anything like them exists today—to put together requirements for the whole range of future software engineering mechanisms and methods, to then formulate a plan to design and implement the imagined languages and tools, and then build practices, theories, and models for software engineering based on these technologies and capabilities.

REFERENCES

- [1] Pollack, W., *et al.* *Ultra Large Scale Systems: The Software Challenge of the Future*. The Software Engineering Institute. http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf. 2006.
- [2] Gamma, E., *et al.* *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1994.
- [3] Davis, S. & Kiczales, G. "Registration-Based Language Abstractions." *Proceedings of the 25th Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Onward! 2010, Reno-Tahoe, 2010.
- [4] Lehman, J.; Stanley, K. "Exploiting Open-Endedness to Solve Problems Through the Search for Novelty." *Proceedings of the 11th International Conference on Artificial Life (ALIFE XI)*. MIT Press. 2008.