Software Engineering in an Uncertain World

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
+1 412 268-5056
garlan@cs.cmu.edu

ABSTRACT
In this paper, we argue that the reality of today's software systems requires us to consider uncertainty as a first-class concern in the design, implementation, and deployment of those systems. We further argue that this induces a paradigm shift, and a number of research challenges that must be addressed.

Categories and Subject Descriptors
D.2.2 [**Design Tools and Techniques**]

General Terms:
Design, Languages, Theory

Keywords
software engineering, software design, software architecture. uncertainty

1. INTRODUCTION
Software Engineering is founded on a computational myth that no longer fully serves its purpose: that the computational environment is predictable and in principle fully specifiable, and that the systems that compute in those environments can in principle be engineered so that they are trouble-free. We act with confidence that a statement like "x := 1" will change the current value of x to 1. We assume that with sufficient effort we can assure ourselves that a given body of code will produce correct results when invoked on arguments satisfying appropriate preconditions.

To be fair, we have known all along that this is an illusion. The machine might crash in the middle of an assignment. We may not have the resources to establish program correctness. A procedure call might fail in a variety of unpredictable ways if that call is made on an object residing on a different host.

However, we do our best to maintain the illusion of certainty, marginalizing errors and unpredictable behavior so that we can focus on the "normal" case, and doing our best to remove any defects during development so as to guarantee that nothing bad will happen during operation. And, until recently, this approach has served us well. It has given us a solid footing for an engineering discipline of software, that has worked reasonably well in a world of desktop applications, code produced by a single organization, and appropriate investments in development-time activities (requirements elicitation, design, analysis and testing, etc.).

Unfortunately, the world no longer fits that profile. Systems are increasingly composed of parts, frameworks, and layers built by many organizations over which we may have little control and which have failure modes that are not fully specified or even understood a priori. Computations increasingly must run in environments in which resources (bandwidth, CPU, screen space, power) may have radical variability, particularly in the presence of mobility. Systems increasingly require cooperation from human users or operators (for example, in achieving security and privacy goals), who may not do what they are expected to do. Qualities such as performance, security, availability, and rapid time to market increasingly trump the desire for correct computation.

To deal with this new reality I argue that it is time to embrace uncertainty as a first class entity, recognizing that in a world where we cannot hope to achieve perfection, we must rethink many of the ways in which we conceive, engineer, and validate our software-based systems. In doing so, we have an opportunity to learn from related disciplines, which have done so for decades, adapting their techniques to mainstream software engineering.

2. SOURCES OF UNCERTAINTY
Let us consider some the sources of uncertainty that are the reality of today's computational world.

- *Humans in the loop.*. Systems increasingly rely on correct human behavior to achieve their goals. For example, system administrators must configure a system properly and adjust operating parameters over time. End users of a system must follow certain procedures, such as changing passwords from time to time, setting appropriate privacy settings, or installing system upgrades on their desktops. And yet humans are clearly not fully controllable or even predictable. While this has always been true, there has been a qualitative shift in the nature of the user community, as software systems have become essential to conducting virtually all of the daily activities that humans engage in.
- *Learning.* Increasingly systems are incorporating machine learning as part of their functionality. This can take many forms, from simple dynamically adaptive menus, to complex cognitive assistants that learn how to help the user with difficult tasks [2]. Most learning-based systems rely on past ob-

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

servations and statistical regularity to determine observations – leading to systems whose behavior is uncertain.

- *Mobility.* Applications today run on a variety of platforms, many of which are mobile: phones, notepads, cars. Resource availability is not only radically different on these platforms, but also can change dynamically and frequently.
- *Cyber-physical systems:* More and more systems involve complex interactions between software and physical elements: building energy management systems, building security systems, medical devices, cars, energy distribution, to name a few. While a large body of knowledge exists about properties and models of physical devices, these are rarely considered by software developers. Such systems are inherently uncertain, and typically require some form of closed loop control in which dynamic observations of the physical environment are required before computations can proceed.
- *Rapid evolution:* The ability to manage the evolution of software systems has been at the heart of software engineering from its earliest days. However, what has changed recently is the pace of that evolution and the context in which it must occur. Today systems must adapt rapidly to new requirements, technologies, regulations, and markets. This introduces new forms of uncertainty about the future state of a system's architecture, interfaces, platforms, and external peers. Moreover, for an increasingly large number of systems, evolution must occur without disruption of system availability, often requiring upgrades in the presence of large-scale distributed platforms, where inherent uncertainty makes it extremely difficult to plan and execute system-wide modifications.

3. IMPLICATIONS

What are the implications of developing software in a world where uncertainty is not just a curious, and second-order, fact of life, but the elephant in the room? We argue that we must embrace uncertainty within the engineering discipline of software engineering, designing systems so that uncertainty becomes a first-class concern that is dealt with systematically. In particular, we argue that this causes a shift in perspective along several important dimensions.

- *From correctness to utility:* In an uncertain world, correctness can no longer be the end goal. Instead we must be concerned with something much weaker, but much more useful: utility. The key question is how can we maximize the value of the system to users while minimizing costs to the providers of that system? Utility naturally accommodates uncertainty, since it allows us to consider *expected value.* And it allows software designers to trade off multiple dimensions, for example, sacrificing correctness in favor of performance. (Consider the use of a cached value that can be delivered quickly, but may, with some probability, be out-of-date.)
- *From open-loop to closed-loop:* Traditional software systems run open-loop: we create them and then deploy them, replacing them with new systems if they are not behaving correctly. In a closed-loop system run-time behavior is monitored and the system is adapted dynamically to improve its behavior. Closed-loop systems are appropriate for handling uncertainty because they allow systems to respond dynamically to unexpected faults, resource variability, and changing requirements. For such systems (sometimes termed autonomic or self-adaptive) the engineering challenge shifts to the run-time mechanisms that must be put into place in order detect problems and adapt a system to correct them.
- *From precise to approximate:* If we cannot completely determine system behavior or guarantee correct behavior in advance, we must find ways to make sure that systems work "well enough." This leads to new ways to think about systems in terms of approximation, but also requires that we find ways to bound those approximations. Today, for example, we use non-determinism to approximate system behavior. However, non-determinism is far too open ended. For example, designing a reliable system using servers that can fail non-deterministically is much different than designing one in which servers fail on average .001% of the time.

To give a concrete example of a system in which these paradigm shifts are evident, consider the architecture of the Google File System [3]. Having decided to use stock server and storage hardware (to minimize cost and to support scalability), GFS adopts a scheme in which a master control component monitors available servers to detect when servers are no longer functioning properly. The master control is then responsible for removing faulty servers dynamically, and balancing the information storage and retrieval functions among the correctly functioning servers.

- *From correctness to utility:* Is the GFS fault management scheme "correct" in terms of providing continuous availability? No: if all the servers crashed simultaneously Google would go down. But the likelihood of such a thing happening is so small that the system has high expected availability at relatively low cost, and hence high utility.
- *From open-loop to closed-loop:* The master control acts as a controller adapting the system as necessary to meet performance and availability objectives.
- *From precise to approximate:* Is it possible to predict how many servers are available at a given time? No. But it is possible to quantify expected availability given probabilities of server faults.

4. RESEARCH CHALLENGES

These perspective shifts induce a number of software engineering research challenges, many of which can be informed by existing research in software engineering and areas outside our field.

- *Design methods:* How can we incorporate uncertainty into our system designs in a way that allows us reason about it as a first class entity, and make principled tradeoffs that are informed by that uncertainty? For example, we might decide to use a server whose uncertainty is low, but performance is high. How can we determine system utility based on knowledge of uncertainty? Relevant work includes economic models (e.g., options theory), game theory, and operations research.
- *Resilient systems:* How can we engineer adaptive systems that provide appropriate behavior in the presence of unpredictable faults, resource variability and changing requirements? Are there ways to compose adaptation modules to

achieve overall goals? Relevant work includes control theory, context-aware computing, and autonomic computing systems [6]. In addition research on robotics planning and machine learning have had to cope with uncertainty since their inception.

- *Formal methods and tools:* What kinds of methods and tools will permit us to reason about systems in the presence of uncertainty. This includes the ability to:

  (a) Reason about the properties of adaptation strategies to determine under what conditions they will achieve their desired goals (e.g., reducing request latency) and whether they preserve behavioral invariants of a system (e.g., messages are not lost).

  (b) Refine specifications as more information becomes available. For example, at early stages of design we may only know that servers can fail, but not at what rate. As we gain more information, we would like to be able to refine our models so that they preserve previously established properties.

  Relevant work includes models for probabilistic behavior (e.g., Markov chains), logics for reasoning about properties of systems under uncertainty (e.g., work by MacIver [4]), and tools for checking properties of probabilistic systems (e.g., Prism [5]),.

- Learning: How can we incorporate machine learning into our systems while retaining the ability to reason about system behavior? Learning systems based on statistical models are notoriously bad at providing any kind of guarantees. Is there a way to bound the behavior of such systems so that we can guarantee that their use is restricted to an envelope of reasonable behavior? Relevant work is being carried out in the machine learning where episodic and statistical approaches can coexist [1].

5. EXAMPLE

As an example of research that touches on many of these areas, we offer as one data point recent work on designing languages for specifying self-adaptation strategies. The Rainbow Project at Carnegie Mellon University uses closed-loop control to monitor a system dynamically and adapt it when opportunities to improve its behavior are detected [6]. A key component of this work has been the development of Stitch, a language for specifying adaptations [7][8]. Stitch views an adaptation as a kind of decision tree of conditions and actions. It is unique insofar as it combines notions of control theory (such as settling time), utility theory (to evaluate the expected impact of a strategy and to select an appropriate one), learning (to improve its estimates of uncertain values over time and to predict the probability that an adaptation strategy will accomplish its goal), and formal semantics based on Markov Decision Processes.

6. CONCLUSION

Uncertainty is a fact of today's software-based systems, and is becoming increasingly important. We have argued that this leads to some fundamental paradigm shifts in software engineering. These in turn engender a set of research challenges for the field, centered on the observation that rather than behaving as if uncertainty doesn't exist, or relegating it to a footnote on normal behavior, we should find ways to bring it into our software engineering methods, tools, notations, and thinking as a first-class citizen. We offered up the Google File System as an example of a modern system whose design cannot be appreciated without understanding the role of uncertainty, and the Stitch language as an example of a programming language in which management of uncertainty is a central focus.

7. REFERENCES

[1] Steinfeld, A., Bennet, R., Cunningham, K., Lahut, M., Quinones, P., Wexler, D., Siewiorek, D. P., Hayes, J., Cohen, P., Fitzgerald, J., Hansson, O., Pool, M., Drummond, M. Evaluation of an Integrated Multi-Task Machine Learning System with Humans in the Loop. In Proceedings of *NIST Performance Metrics for Intelligent Systems Workshop (PerMIS),* Jan, 2007.

[2] Garlan, D. and Schmerl, B. The RADAR Architecture for Personal Cognitive Assistance. In *International Journal of Software Engineering and Knowledge Engineering*, Vol. 17(2), April 2007.

[3] Ghemawat, S., Gobioff, H., and Leung, S. The Google File System. *Proceeding of the 19th ACM Symposium on Operating Systems Principles*,Lake George, NY, October, 2003..

[4] McIver,A.. Quantitative Refinement *and* Model Checking for the Analysis of Probabilistic Systems. In *Formal Methods 2006*, Springer, LNCS 4085, pages 131-146, August 2006.

[5] Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.. "PRISM: A Tool for Automatic Verification of Probabilistic Systems". *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of Lecture Notes in Computer Science, pages 441-444, Springer, 2006.

[6] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. "Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure," *IEEE Computer* Volume 37, Number 10, October 2004.

[7] Shang-Wen Cheng and David Garlan. Handling Uncertainty in Autonomic Systems. In Proceedings of the International Workshop on Living with Uncertainties (IWLU'07), November 2007.

[8] Cheng, S. Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation. PhD. Thesis. CMU-ISR-08-113. May 2008