Bits of History, Challenges for the Future and Autonomic Computing Technology

Hausi A. Müller
Department of Computer Science
University of Victoria, Canada
hausi@cs.uvic.ca

Abstract

Over the past fifteen years, the software reverse engineering community has produced many software engineering methods, tools, and techniques that have had significant impact in the software industry. After a research area has evolved for 10-20 years, it can easily fade away due to narrow focus, overgrazing, or lack of impact. Trying to assess the impact of various approaches and results in a research area is difficult—but worthwhile. Taking a step back and looking at a research area from new perspectives is probably easier and can be invigorating. The lessons learned from such exercises may result in new research challenges, foster cross-fertilization among research areas, and shape the focus of the research communities.

Inspired by several recent studies that assess the field of software engineering as a whole to define research agendas and funding policies, I discuss several new perspectives on the problem of continuous software evolution that will hopefully inspire the reverse engineering community. I then advocate that we need to push monitoring of evolving systems to unprecedented levels to be able to observe and possibly orchestrate their continuous evolution in a complex and changing environment. I then suggest to instrument evolving software-intensive systems with autonomic elements, using reverse engineering techniques, to enhance their monitoring and assessment capabilities.

1. Introduction

Back in 1999 Truex et al. suggested in a thought-provoking CACM paper that the problem with evolving software systems may lie in an incorrect goal set that we all have accepted [43]. The assumption that “software systems should support organizational stability and structure, should be low maintenance, and should strive for high degrees of user acceptance” might be flawed. They suggest an alternate view that assumes “software systems should be under constant development, can never be fully specified, and are subject to constant adjustment and adaptation” [43].

Having investigated software systems and their engineering methods for almost 30 years, I have come to realize that their alternate, continuous evolution view is more accurate than the traditional view. This is good news for our reverse engineering research community since this view guarantees research problems for years to come. The bad news is that most software engineering textbooks will have to be rewritten.

Several impressive studies conducted over the past year seem to confirm that this notion of continuous evolution is taking hold [5, 7, 29]. In particular, a Carnegie Mellon Software Engineering Institute (SEI) study suggests that traditional top-down engineering approaches are insufficient to tackle the complexity and evolution problems inherent in decentralized, continually evolving software. The key message I would like to communicate in this paper is that we have to push monitoring of evolving systems to unprecedented levels to be able to observe and possibly orchestrate their continuous evolution in a complex and changing environment. Moreover, autonomic computing technology has matured to a stage where it can be used to reduce complexity and solve evolution problems of software-intensive systems. Finally, the reverse engineering community—with all its methods and tools—is ideally positioned to instrument long-lived software systems with autonomic elements.

The remainder of this paper is organized as follows. First I pay tribute to the success stories of our field. In Sections 3, 4, and 5 I attempt to characterize the problem of continuous software evolution using different perspectives. Section 6 provides a brief
overview of autonomic computing technology and Section 7 outlines how this technology can be used to instrument software systems for continuous evolution.

2. Success stories

The field of software reverse engineering and closely related fields, such as program comprehension or software analysis, have enjoyed many successes over the past fifteen years—in academia as well as industry. Many theories, methods, tools, and techniques have been developed to ease and facilitate the understanding, maintenance, evolution, transformation, migration, and reengineering of software-intensive systems.

We have become proficient in modeling and extracting artifacts and dependencies, exploring software systems [28, 35], composing and clustering subsystem structures [43], uncovering architectural styles [30], computing change impact, synthesizing dynamic dependencies [38], recognizing patterns, assigning concepts, redocumenting programs [13, 40], recovering objects and components [22], refactoring code [25, 39], sniffing out code smells, detecting clones [21], slicing and dicing [23], mining versions and repositories [9, 11], finding concern graphs [31], and recording cross-cutting and multi-dimensional concerns.

Most reverse engineering environments have several standard tool components [13]: extractors, analyzers, visualizers, and repositories. Many data, control, and presentation integration mechanisms have been developed to facilitate interoperability among these tool components [26]. Important tooling research issues include tool requirements and evaluation [18, 34, 45], meta-models, schemas, and exchange formats to capture reverse engineering knowledge at different levels of abstraction [12], and visualization strategies and mechanisms which cater to different types of users and tasks [35, 37].

The success stories of reverse engineering research are reported in the premier software engineering conferences (i.e., ICSE, ICSM, CSMR, FSE, and ESEC) as well as specialized conferences (e.g., WCRE, ICPC (formerly IWPC and WPC), SCAM, PASTE, MSR, VISSOFT, ATEM, IWPSE, and WSE). The vitality of the field is amply exhibited at these more specialized events which shape and focus the field. From the beginning, industrial case studies and legacy subject systems were an essential and integral component of reverse engineering research. In recent years there has been a high expectation in the research community that tools should be subject to some sort of evaluation [35, 37, 44]. Case studies of industrial systems are often used as a mechanism to demonstrate that tools are effective in helping developers reverse engineer systems.

Today we have a much better arsenal of analysis and transformation methods and tools than 15 years ago to deal with mass software. Examples which demonstrate the effectiveness of this arsenal include Y2K or Euro conversions, migration to object-oriented or network-centric platforms, injection of COTS components and user interface technologies, or transformation to Service-Oriented Architectures (SOA).

Nevertheless, the complexity and continuous evolution challenges we face today seem to grow and overwhelm even the most advanced software reverse engineering capabilities. Yes, we can parse millions of lines of code and extract all kinds of artifacts, and yet we are surprised when requirements change or evolve. Of course we should continue to work on advancing and perfecting the current methods and tools. But I think a reorientation is needed to attack the problems posed by continuous evolution.

3. Continuous evolution of systems

Legacy systems typically refer to large, complex systems that have evolved to a state where they significantly resist further modification and evolution [3, 4]. Over the past decade, the life of many of these cash cows of the software industry has been extended by migrating them to modern platforms [6] (e.g., object-oriented programming language [20, 24], Web-based user interface, 3-tier Web service [41], or product line architecture). Recently, in part due to the proliferation of the Web, those same systems seem to be morphing again into highly configurable systems of systems.

According to a recent SEI study, systems of systems are likely to evolve into Ultra-Large-Scale (ULS) Systems [29]. One of the characteristics of ULS systems is that they cannot be fully engineered in a top down rational manner. For example, the protocols of the Internet are engineered, but not the Web as a whole. The SEI study uses the metaphor of a city to illustrate the characteristics of ULS systems. A city, such as Rome, consists of layers built by many individuals over long periods of time. A recent National Geographic article attests to this layered construction by exploring Rome’s basement via its sewers and heralds it as the “world’s largest undiscovered museum” [2]. The form of the city, as opposed to individual buildings and infrastructure
components, is not specified by requirements, but loosely coordinated and regulated by zoning laws, building codes, or economic incentives. Rome has been constructed, repaired, improved, and changed over centuries and never ceased to function. Individual buildings and pieces of infrastructure (e.g., sewer or water supply) are engineered, but not the city as a whole. The authors of the SEI study argue convincingly that ULS systems will not simply be bigger systems, but rather interdependent, dynamic and competing networks of software-intensive systems, people, policies, cultures, and economics [29].

Such systems are no longer built by satisfying requirements through traditional, top-down engineering, but rather by satisfying requirements by regulating complex, decentralized, interdependent subsystems [29]. In traditional software engineering, functional and non-functional requirements are hard-wired into the systems and thus need not be monitored for continuous satisfaction. In the future, monitoring of requirements will likely be the norm rather than the exception. In fact, monitoring must be pushed to unprecedented levels to be able to observe and orchestrate interdependent and competing eco-components in a changing environment.

4. New perspectives are needed

Over the past year, several impressive studies have been conducted to assess the field of software engineering as a whole, to define research agendas, and to change research funding policies. The lessons learned from such studies may lead to new research challenges for research areas such as reverse engineering, foster cross-fertilization among related areas, and shape the focus of the respective research communities.

In his seminal ICSE 2006 keynote address, Boehm characterized software engineering with a sequence of theses and antitheses for every decade from 1950 to 2020 [5]. He argued that “the ability of organizations and their products, systems, and services to compete, adapt, and survive will depend increasingly on software and on the ability to integrate related software-intensive systems into systems of systems.”

In a recent manifest, which originated at a Dagstuhl seminar entitled “Challenges for Software Engineering Research,” a group of leading German Software Engineering Professors call for a significant expansion of the number of software engineering groups in their country [7]. This call for action stems from the realization that software is of critical and strategic importance. Software systems are central to our lives and the machines and products traditionally built by mechanical, electrical, civil, and process engineers contain today an inordinate amount of software. Also of note is that the first bullet under technical research topics in this manifest refers to “reverse engineering and migration to modern platforms.”

The one-year long SEI study mentioned above, led by Northrop, goes a step further and argues that systems of systems are likely to evolve into Ultra-Large-Scale (ULS) socio-technical ecosystems [29]. The SEI team posits that managing such ULS ecosystems requires a radically new perspective with respect to design and evolution, orchestration and control, as well as monitoring and assessment.

These studies have definitely influenced my own view on why dealing with legacy systems does not seem to get any easier despite all the significant progress in recent years.

From a researcher’s perspective, one of the obvious reasons is the obstacles encountered in adopting software engineering research results—methods and tools—in industrial practice. We have advocated Adoption-Centric Software Engineering (ACSE) to overcome some of these obstacles, but there are still many opportunities for improvement in this realm [1].

A common realization in the software industry—the software developer’s perspective—is that the increasing complexity of computing systems is overwhelming the capabilities of software developers and system administrators to design, evaluate, integrate, and manage these systems. Thus, major software and system vendors are concluding that the only viable long-term solution is to create computing systems that manage themselves. Every major software vendor has launched an initiative to attack the software complexity problem: Autonomic Computing (IBM), Adaptive Enterprise (HP), Dynamic Systems Initiative (Microsoft), Proactive Computing (Intel), or Harmonious Computing (Hitachi) [27].

Finally, a third perspective might be an end-user programming perspective. End-user programming using scripting languages grafted on top of many software applications ranging from Web browsers to spreadsheets has become pervasive. Unfortunately scripts are plagued with quality and reliability problems. Providing support to these end-user programmers to monitor and assess the quality of their programs is a worthwhile research goal. Burnett et al. introduced the notion of end-user software engineering, a strategy that allows spreadsheet end users to enhance the dependability of their scripts using quality control devices [8]. Shaw argues that the dependability game should change. She makes the case for a value-based software engineering approach.
where “an appropriate strategy should strike a balance between preventing as many problems as is cost-effective, then repairing any others that occur” [33].

5. History of complexity problems

Today, computing systems include very complex infrastructure and operate in complex heterogeneous environments. With the proliferation of hand-held devices, the ever-expanding spectrum of users, and the emergence of the Web information economy, computing vendors have a tough time providing infrastructure to address all the needs of users, devices and applications.

Web services with Service-Oriented Architectures (SOAs) as their core technology have solved many problems, but also raised numerous complexity issues. The infrastructure of today’s information system is also highly configurable with literally hundreds of components and thousands of tuning parameters. Thus, configuring and optimizing an information system consisting of an application server with over 100 tunable configuration parameters, a web server with 30 parameters, and messaging with 20 parameters is very complex for humans and costly to install and maintain [10, 15].

In a 2004 Economist article, Kluth investigates how other industrial sectors successfully dealt with complexity [19]. He and others have argued that for a technology to be truly successful, its complexity has to disappear. He illustrates his arguments with many examples including the automobile and electricity markets. Only mechanics were able to operate early automobiles successfully. In the early 20th century, companies needed a position of Vice President of Electricity to deal with power generation and consumption issues. In both cases, the respective industries managed to reduce the need of human expertise and simplify the usage of the underlying technology. However, usage simplicity comes with an increased complexity of the overall system complexity (e.g., what is under the hood). Basically for every mouse click or return we take out of the user experience, 20 things have to happen in the software behind the scenes. Given this historical perspective and predictable path of technology evolution, maybe there is hope for the information technology sector [19].

6. A blueprint for autonomic computing

A software component that operates on its own or with a minimum of human interference according to a set of rules is called autonomic. The term autonomic derives from the human body’s autonomic nervous system, which controls key functions without conscious awareness or involvement [10, 14].

IBM started the Autonomic Computing initiative in 2001 to build self-managing computing systems to overcome the rapidly growing software complexity problems [10, 14, 17].

IBM researchers have developed several iterations of an architectural blueprint for autonomic systems [16, 17]. An autonomic system consists of a set of autonomic elements that contain and manage resources and deliver services to humans or other autonomic elements. An autonomic element consists of one autonomic manager and one or more managed elements. At the core of an autonomic element is a control loop which integrates the manager with the managed element.

The autonomic manager comprises a manageability interface (i.e., sensors and effectors) and a five-component analysis and planning engine. It is imperative that the information being exchanged across a manageability interface be standardized so that autonomic managers can manipulate other autonomic elements in a uniform manner.

The monitor observes the sensors, filters the data collected from the sensors, and then stores the distilled data in the knowledge base. The analysis engine compares the collected data against the desired sensor values also stored in the knowledge base. The planning engine devises strategies to correct the trends identified by the planning engine. The execution engine finally adjusts parameters of the managed element by means of effectors and stores the affected values in the knowledge base.

An autonomic element manages its own internal state and its interactions with its environment (i.e., other autonomic elements). An element’s internal behavior and its relationships with other elements are driven by goals and policies the designers have built into the system.

An autonomic system, built using self-managing autonomic elements, can self-configure at run-time to meet changing operating environments, self-tune to optimize its performance, self-heal when it encounters unexpected obstacles during its operation, and self-protect itself from malicious attacks.

A good starting point for experimenting with autonomic elements is the autonomic computing toolkit available under IBM developerWorks [14, 16].

IBM has been exceedingly successful in rallying the research community behind their autonomic computing initiative [27]. Several conferences and workshops have emerged including IEEE International Conference on Autonomic Computing (ICAC); ACM
Workshop on Self-managed Systems (WOSS); ACM Workshop on Design and Evolution of Autonomic Systems (DEAS); IEEE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS); Autonomic Computing Workshop (AMS); Conference on Human Impact and Application of Autonomic Computing Systems (CHIACS); Autonomic Applications Workshop (AAW); Engineering of Autonomic Systems (EaAS); and Workshop on Software Architecture for Dependable Systems (WADS).

7. Instrumenting software systems

IBM has delivered in simplifying data base management systems (e.g., DB2) and configuration managers (e.g., Configuration Advisor or the Tivoli Risk Manager) by instrumenting them with autonomic elements embodying self-configuring, self-optimizing and self-healing capabilities. Moreover, IBM has woven more than 475 autonomic features into more than 75 products [15].

Hitherto, most developers did not instrument their software with sensors and effectors to observe its hard-wired requirements. One way to break out of this mold is to monitor assertions and invariants, which are typically comments in the source text, using autonomic elements. The frequency of raised exceptions or runtime check violations could be computed and used to assess program deterioration. When a hard-wired array gets full, a monitor could unobtrusively adjust the table size in the program and then recompile, reinstall and redeploy the program. Critical regression tests could be regularly performed to observe the satisfaction of requirements.

While we are accustomed to common need monitors such as command line completion, default form filling, performance profiling, or even checking liveness properties, we have difficulties imagining how to monitor and assess the satisfaction of requirements in a long-lived software-intensive system of systems. However, if we start instrumenting systems in a bottom-up manner using autonomic control loops, then I believe we can build structures of autonomic elements that can monitor and regulate the functional or non-functional requirements of interdependent and competing eco-components in a changing environment over long periods of time. For example, I can imagine that self-managing elements will be able to monitor and control the brittleness of legacy systems, provide automatic updates to evolve installed software, adapt safety-critical systems without halting them, immunize computers automatically against malware, facilitate enterprise integration with self-managing integration mechanisms, document architectural drift by equipping systems with architecture analysis frameworks, or keep the values of a quality attribute within a desired range.

The challenge therefore for the reverse engineering community is to instrument software systems with manageability endpoints (i.e., sensors and effectors) using the reverse engineering and transformation technology developed over the past 15 years.

8. Conclusions

While the reverse engineering roadmap we laid out for the Future of Software Engineering Track at ICSE 2000 [26] is still valid, our perspective has changed considerably.

On the one hand, dealing with software-intensive ecosystems seems rather daunting when we just became comfortable reverse engineering stovepipe legacy systems. On the other hand, the need to regulate large, complex, decentralized systems by monitoring and tuning the satisfaction of requirements is imminent and possibly a new raison d’être for the reverse engineering community. We need numerous sensors, monitors, analysis/planning engines, and effectors to be able to observe and control independent and competing organisms in a dynamic and changing environment.

Appreciating the need to reduce complexity or to enhance the quality of end-user programming might be more palatable at the moment. I submit, however, that autonomic computing technology can be employed to attack all of these problems. The reverse engineering community—with all its methods and tools—is of course ideally positioned to instrument long-lived software systems with autonomic elements.

As a final thought, I propose to teach the notion of a control loop, together with the concept of an autonomic element, early in computer science and software engineering curricula. In contrast to engineering disciplines, computing science programs seem to have neglected the control loop. Given the above discussion, it might be high time to give it more prominence. Over a decade ago, Shaw compared a software design method based on process control to an object-oriented design method [32]. Not surprisingly, the process control pattern described in that paper resembles an autonomic element.
Acknowledgements

I would like to thank Holger M. Kienle, Scott R. Tilley and Ken Wong for comments on a draft of this paper.

My work has been supported by National Sciences and Engineering Council (NSERC) of Canada, Canadian Consortium for Software Engineering Research (CSER), IBM Toronto Center for Advanced Studies, IBM Corporation, Carnegie Mellon Software Engineering Institute, and the University of Victoria, British Columbia, Canada.

References


