Evolving Evolution

Steven P. Reiss
Brown University
Providence, RI 02912
401-863-7641, spr@cs.brown.edu

Abstract

Software is changing and software evolution is going to change with it. In considering software and the problems of software evolution today we make the tacit assumption that we control the software and hence can control its evolution. Current trends point to a world where we only control a small fraction of our own software and the remainder evolves in unpredictable and uncontrolled ways. Current work in addressing evolution is addressing yesterday's problems. What we need to prepare ourselves for the coming problems are techniques that can cope with uncontrolled evolution. In this position paper we point out several of the problems that need to be addressed and hint at possible techniques for addressing them.

1. Introduction

Software evolution deals with the changes in a software system over time. Today our software systems are composed of potentially large numbers of different types of artifacts. The artifacts represent different portions of the software, ranging from requirements, specifications and design at the front end to source code, test cases, and documentation on the implementation side. As software systems grow and change, all of these artifacts evolve at different rates and in different ways.

Such evolution is one of the primary causes for the many problems associated with software systems. The lack of understandable design, incoherent source code, poor internal documentation, and incomplete test cases, all resulting from the differing evolution, lead to unreliable and untrustworthy software. Incomplete or inaccurate external document leads to user confusion and frustration. Designs and implementations that don't anticipate evolution are difficult to change and adapt to changing environments and platforms. Inconsistent artifacts yield more and more complex software, software that eventually becomes so complex it has to be scrapped and completely rewritten to remain viable.

These problems have led us as a community to develop a variety of solutions aimed at addressing and managing software evolution. We as a community attempt to understand the process of software evolution so we can anticipate and adapt to it rather than forcing it to adapt to us. This is seen in agile programming, in production-line environments, and in feature-based or aspect-oriented programming. We try to provide ways of controlling software evolution by developing tools that can check the evolving software and ensure it remains consistent. Here we have developed regression testing, tools for statically or dynamically checking software for common errors such as memory usage, and more recently tools for statically checking software against its specifications. We also develop tools and techniques for physically doing software evolution. This can be seen in the various refactoring tools that have become part of today's programming environments such as those found in Eclipse. We have also worked on high-level languages that let us work in simpler terms in terms of specifications or designs and have the code automatically generated, thereby simplifying the artifacts that need to evolve.

Our own work in recent times has taken a different tact. In addition to all the above work, we felt that one needed to address the problem of inconsistent evolution, where the different artifacts composing the software system evolve at different rates and in different ways. Rather than attempting to enforce consistency or even a methodology, we provided a tool, CLIME, that simply checked that the artifacts were and remained consistent with one another as they evolved and identified inconsistencies to the programmer as they arose [11].

However, almost all this work, ours included, is already out of date before we complete it. We are solving yesterday's problems on yesterday's software for yesterday's systems. Tomorrow's software will be different and have a very different evolutionary framework. We need to start working on how we deal with the new problems that will arise now, not ten years from now when they overwhelm us. We need to evolve our thinking on evolution.

2. Uncontrollable Evolution

Most of the research involving software evolution and most of the frameworks that have been built to deal with it make the tacit assumption that the programmers control the software and the software artifacts and

hence have control over the evolution of the software. This is what makes these techniques viable — there is hope that they can provide solutions and tools that will successfully harness evolution. Unfortunately, while this assumption is mostly true today, it will no longer be true in the future.

The concept of a program is changing from a local, self-contained object into an Internet-scale, pervasive, self-organizing, omnipresent entity. This can be seen in a number of current trends including web services, grid computing, peer-to-peer computing, autonomic software [4], the open source movement, and faster, pervasive, more-reliable networking.

These trends are converging. Programs are written using a multitude of web services some of which use other web services. These web services are written by different, often anonymous programmers and change with little or no notice. Grid-based applications such as today's database systems run on any available machine and find new computational resources as needed. Data is shared across the Internet both through web services and through peer-to-peer connections. Instant messaging, SETI, Gnutella [2], Napster, and electric sheep [3] are examples of current Internet-scale applications, with more to come.

Dealing with this new reality will require us to change the way we think about programs and programming. We can't continue to think of a program as a self-contained entity with only local effects that we control. Instead, we need to work in terms of a global system where we do not control most of the components or interactions. We need a means for programming in a world where software systems will be built mainly from components designed, developed, maintained, and modified by different people, on machines that are not under the control of the software developers or designers, and where the components themselves evolve outside of the control of the software system.

Programmers have always had to deal with outside factors. However, most of these factors were somewhat under their control. Compilers, loaders, operating systems, and tools all change over time and systems need to adapt to these changes. However, programmers typically could choose when to upgrade the operating system or when to change compilers.

Uncontrolled evolution is more difficult. The FIELD system integrated a variety of programming tools including the debugger, editors, make, and rcs using a message-passing mechanism and tool wrappers [9,10]. Here we encountered one of our first examples of external evolution. The wrapper for the debugger was the most problematic. It operated by parsing translating FIELD commands into debugger (gdb or dbx) commands and then parsing the debugger output to determine what was going on. Unfortunately the debugger output syntax was not considered part of its interface by its developers and hence changed frequently and significantly. The result was that every time a new version of either debugger became available, we had to make significant changes to the wrapper. Moreover, because we had to work on multiple platforms, these changes tended to occur relatively frequently.

A more recent experience is even more telling. For one of our visualizations we needed to get the OpenDirectory classification for a web page. The Google web service provides a programmatic way of making Google web searches with the results returned as structures. Part of this structure is the OpenDirectory category for the found page. By searching for the particular page we were interested in, we were able to get a fast and accurate classification. However, Google changed their underlying framework (but not the web service) so that the category was no longer computed for recent pages and hence, suddenly we found that pages that previously had a category, no longer did. Our application ceased to work.

These simple examples are harbingers of what is coming as we move to a world where we are dependent on web services, libraries, shared data and databases, and global computing.

3. TAIGA

In order to deal with these new realities, we need to develop new approaches to software and to software evolution. As an example of an approach that might work, we are currently developing a prototype framework, TAIGA. This framework is designed to provide programming support for a world where web services and other external components abound, where shared data is the rule rather than the exception, and where programs run wherever there are available resources.

The prototype is based on a unique and comprehensive component model. This model is a metaprogramming model where we provide a language for defining the interface to which users will code and a separate language for defining how a particular piece of code, be it a web service, a library, a JavaBean, or a CORBA object, matches that interface. We then provide facilities for automatically and dynamically binding an interface to an appropriate implementation. The metaprogramming approach ensures us portability and allows the system to use existing components such as web services without modification. The model allows the definition of security, privacy, and recovery properties. Finally, the semantics of an interface is defined primarily by a set of test cases that is included as part of the interface. While the current prototype works only with Java, there is no inherent reason why the technology would not extend to be language independent.

To support this model, we use have implemented a peer-to-peer system on top of JXTA [5]. This peer-to-

peer backbone supports the sharing of interfaces and implementations and, at the same time, provides a global file system and shared data facilities. Interfaces and implementations are registered and assigned unique version numbers. When an application first tries to use an interface, the peer-to-peer system finds potential implementations and chooses the most appropriate one given the application's constraints. The binding occurs at run time and can be changed dynamically to handle fault recovery, broken network connections, or even dynamic upgrades of the components. This approach to dynamic binding is more general and broader-based than previous approaches such as [7,12].

The prototype also provides an initial approach to global data sharing. It uses a global name space where each computer has a unique hierarchical name. On top of this it supports a simple global file system with file sharing and the ability to create and write files. It also supports shared tuple spaces based on Linda [1] and shared SQL databases.

4. Component Semantics

Our work on TAIGA points out one way that we might be able to get a handle on software evolution where much of the software is out of our control. The key here is to have a component model where the semantics of the component are included in its interface.

TAIGA uses a separate interface definition language to define what is essentially a Java package. This language is designed to work in an open-source environment where implementations are done and controlled independently of the interfaces. To differentiate our interfaces from Java interfaces, we call ours an *outerface*. Our approach differs from other interface languages such as that of Microsoft ActiveX or CORBA in that *outerfaces define both the syntax and the semantics of the potential calls*. By semantics here we mean not only what the calls do, but also how the implementation addresses issues such as security, privacy, availability, and reliability.

What each call does needs to be expressed in a checkable manner to make the approach practical. One approach is to express the semantics using a mathematical language such as Z [13] or Larch [6] or a high-level specification language. A more widely used approach is to use contracts as introduced in Eiffel [8]. Contracts attempt to constrain the behavior of a process by defining what inputs are allowed and how the outputs are related. They are typically expressed in the form of preconditions and postconditions on methods or as conditions on a class. Another solution can be found in the agile or extreme programming approach to development where test cases are developed first and the implementation is tested continually. The test cases provide the developers with a good sense of what the implementation should do and ensure that the resultant implementation works correctly, at least for the circumstances it was envisioned for.

Our approach uses a combination of these. The main semantics of the outerface are specified by a set of test cases defining the implementation's behavior. The system guarantees that any implementation passes all the test cases of the outerface. In addition, the outerface can include preconditions and postconditions on methods and general constraints on the behavior of an outerface class. Going beyond this, TAIGA will provide the means for expressing security and privacy properties required from the implementation, an economic model that can be used to choose among competing implementations, a domain of accessibility, and a fault recovery model that allows dynamic switching of implementations.

Once we have a way of checking implementations against an interface, we have a means for ensuring that changes outside the software system can be controlled. TAIGA supports three different operations. First, a user can register an outerface. This generates a jar file that can be programmed against and makes the outerface available for general use. Second, a user can register an implementation. This associates a web service, a library, or a server component with a set of outerfaces. Here a set of jar files is generated containing the code for internal components and containing the code needed to connect to external components. While implementations can be registered arbitrarily, they will not be used by TAIGA until they are bound. This, the third operation, compares the semantics in the outerface with the code in the implementation, running any test cases, doing static checking as appropriate, and generally matching the specifications. It is only when the system is assured that the implementation is acceptable, that it enables it to be used by applications.

Evolution can be handled in this framework in various ways. As we understand what functionality we want from a web service or other component, we can create test cases for that functionality. TAIGA lets the programmer extend an outerface with additional test cases and then finds implementations that pass not only the original test cases, but the new ones as well. If the existing external components fail the test cases, other implementations will be chosen. As outside components change, their developers will register a new version of them with the system. TAIGA will detect this new version and, if it passes the appropriate semantic tests, will use it.

We have been using TAIGA to develop a visualizer of what people are looking at in their web browsers. This application has three outerfaces, one for recording web pages as they are viewed, one for managing the data files that retain the history, and one for finding the OpenDirectory category of a web page based solely on its URL. We have experienced both types of evolution with the last outerface. First, as our

visualizer changed, we found that we wanted to make sure that certain pages mapped to certain categories and we augmented the test cases for the outerface accordingly. Second, we created several different versions of our own open-directory decision tree implementation and let the application choose the one that was most appropriate.

5. Hidden Evolution

The TAIGA approach doesn't cover everything however. Because we lack control of the external components, they can evolve in ways that are hidden to the rest of the system. The implementation of a web service can change even through its external representation, a WSDL file, stays the same. Alternatively, some aspect of the information provided by the web service can change without affecting the semantics of the service as perceived by the provider. This was the case, for example, when Google started not returning categories on updated pages.

This is a more complex problem that we haven't complete determined how to deal with. One possibility is to have the bindings time out for external services after a given period. The problem here is defining the period so that we don't have to do excessive testing but so that changes can be detected in a timely fashion. Another approach is to do random retesting of bindings where the random interval is chosen to approximate the time-out period. This suffers many of the same problems, but allows the time-out to be changed internally to reflect the past stability of the implementation. A third possibility is to augment the set of preconditions and postconditions associated with the outerface and to use failures of these conditions to trigger retesting.

Such problems, however, are only a small and simple sampling of the many problems we are going to face as we move toward a world of outside components. Imagine these problems when our application calls a web service which calls a web service which calls a web service which calls a web service that has changed in some subtle way. Imagine what happens when the database that one of these web services is using is changed so that the data that is returned is slightly different or so that your requests suddenly take significantly longer than they used to.

Software evolution is evolving away from what we are currently doing toward a world where the software is outside of our control. The challenge of understanding and providing for software evolution today is to provide the tools and techniques that can and will work in the future, not those that would have worked in the past.

Acknowledgements. This work was done with support from the National Science Foundation through grants CCR9988141 and CCR0086057.

6. References

1. Sudhir Ahuja, Nicholas Carriero, and David Gelernter, "Linda and friends," *IEEE Computer* Vol. **19**(8) pp. 26-34 (August 1986).

2. Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker, "Making Gnutella-like P2P systems scalable," *Proc. ACM SIGCOMM 2003*, (Aug 2003).

3. Scott Draves, "Electric Sheep," *http://electricsheep.org*, ().

4. A. Ganek and T. Corbi, "The dawning of the autonomic computing era," *IBM Systems Journal* Vol. **42**(1) pp. 5-18 (2002).

5. L. Gong, "JXTA: a network programming environment," *IEEE Internet Computing* Vol. **5** pp. 88-95 (2001).

6. J. V. Guttag, J. J. Horning, and J. M. Wing, "The Larch family of specification languages," *IEEE Software* Vol. **2**(5) pp. 24-36 (March 1985).

7. Nenad Medvidovic, "On the role of middleware in architecture-based software development," *SEKE '02*, pp. 299-306 (July 2002).

8. Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall (1988).

9. Steven P. Reiss, "Interacting with the FIELD environment," *Software Practice and Experience* Vol. **20**(S1) pp. 89-115 (June 1990).

10. Steven P. Reiss, "Connecting tools using message passing in the FIELD environment," *IEEE Software* Vol. **7**(4) pp. 57-67 (July 1990).

11. Steven P. Reiss, "Constraining software evolution," *International Conference on Software Management*, pp. 162-171 (October 2002).

12. Ran Rinat and Scott Smith, "Modular Internet programming with cells," *Proc. ECOOP 2002, Springer-Verlag LCNS 2374*, (2002).

13. J. B. Wordsworth, *Software Development with Z*, Addison-Wesley (1992).