Appeared in the Proceedings of the International Conference on the Principles of Software Evolution (IWPSE 1).
Kyoto, Japan. April 20-21, 1998.

Decentralized Software Evolution

Peyman Oreizy

Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
+1 714 824 8438
peymano@ics.uci.edu

1 INTRODUCTION

Developers frequently confront a dilemma that may be characterized by the following:

> "90% of the functionality requested by our customer is available in an existing off-the-shelf application, but the missing 10% is vital to the customer. Unfortunately, we cannot customize or adapt the existing application to meet our customer's needs—we have no choice but to buid a custom solution from scratch."

As a result, a relatively small change in functionality necessitates a disproportionately large amount of effort, and curtails the opportunity for software reuse. This undesirable outcome may have been avoided if the off-the-shelf application supported *decentralized software evolution*, which we define as the ability to evolve software independent of the original software vendor.

Decentralized software evolution (hereafter abbreviated DSE) provides a means for third-party software vendors to customize existing applications for particular domains and customers. This capability benefits everyone involved: the original application vendor sells more product since customization constitutes use; the third-party developer delivers a product in less time and with lower cost by reusing software instead of starting from scratch; and the customer receives a higher quality product in less time and with lower cost.

Although reliable, rapid, and cost effective software evolution has been a principal concern of software research since the 1970's, results to date do not directly address DSE. The principles and techniques of software evolution—anticipation of change, separation of concerns, modularity, information hiding [13], object-oriented design [2], mediator-based design [17], adaptive object-oriented design [10], design patterns [7], aspect-oriented design [9], etc.—help *design* evolvable software systems. Unfortunately, the flexibility attained using these techniques is lost when the application is compiled for use. The compilation process solidifies the plasticity of a design, making it exceedingly difficult to accommodate a change that would otherwise be easy to make. The objective of DSE is to preserve the design's plasticity in the deployed system, thereby enabling third-party evolution.

The rest of the paper is organized as follows. Section 2 characterizes DSE within the broader context of software evolution. Section 3 surveys existing techniques for supporting DSE. Section 4 presents our approach to DSE, and section 5 summarizes our experience in applying our approach to several example applications. Section 6 discusses some open issues.

2 SOFTWARE EVOLUTION

Table 1 categorizes common software evolution technologies based on *when* they can be applied and by *whom.* Software can either be evolved by a centralized authority, such as the software vendor (top row), or by a decentralized group, such as multiple independent software vendors (bottom row). Software can also be evolved during the design phase (left column), or after it has been deployed to customers (right column).

Centralized, design-time evolution: A large majority of the available techniques and tools support this category of software evolution. For example, design notations and methods, such as the Booch and Rumbaugh object-oriented methodologies, provide guidelines for system design and diagrammatic notations for design capture. Design tools, such as Rational Rose, automate the diagramming process and provide analysis support. Group communication and collaboration tools, such as e-mail, revision control tools, and configuration management systems, help teams members coordinate and manage software changes.

Decentralized, design-time evolution: Larger teams and geographic distribution differentiate decentralized design-time evolution from its centralized counterpart. The impact of large teams on software engineering environments has been discussed by Perry and Kaiser [14]. They argue that as a project grows above approximately 20 people, the number and complexity of interactions increases. As a consequence, additional rules and mechanisms that enforce cooperation among personnel are needed. Less well understood is the impact of geographic distribution on software development. Fielding and

		Design-time (or pre-deployment) evolution	**Post-deployment evolution**
Who	**Centralized authority (single vendor)**	Design notations, methods, and tools; process system; group communication and collaboration tools; configuration management tools	Release management systems; software patch files
	Decentralized group (multiple independent software vendors)	Design notations, methods, and tools; multi-site process system; wide-area group communication and collaboration tools; distributed configuration management tools	Software plug-ins; scripting languages; developer APIs

(Column heading above the table: **When**)

Table 1: This 2x2 matrix categorizes different techniques used to support software evolution based on *who* can evolve the system and *when* evolution can take place.

Kaiser [5] describe the processes and tools adopted by one particular globally distributed team that develops the Apache Web server. They identify the importance of e-mail communication, archival of e-mail communication (as a means to support group memory), a shared information space accessible by project members, and coordination tools. Cutkosky et. al. [4] report similar experiences using the Internet in the manufacturing domain.

Centralized, post-deployment evolution: Software vendors evolving (i.e., upgrading) their deployed applications is represented by this category. Since evolution is done by a single authority, change conflicts do not arise. As a result, most technologies are concerned with the efficient distribution of upgrades. The most common technology in use today is the binary patch file, which encodes the byte-level changes necessary to upgrade an application to a subsequent release. More sophisticated tools, such as Tivoli's TME/10 [20] and SoftwareDock [8], use software dependency information to guide software upgrades.

Decentralized, post-deployment evolution: Multiple software vendors independently evolving a deployed application is represented by this category. The most popular techniques include software plug-ins, which are used by Netscape's Communicator to support new content types, and scripting languages. There are a host of issues and consequences inherent in supporting this type of evolution. For example, configuration management becomes necessary if conflicts between independently developed software add-ons can occur. Since applications are evolved in-the-field, anomalies may arise from unforeseen interactions between independently developed add-ons. Consequentially, application consistency must be verified whenever the application is modified (i.e., when add-ons are installed or removed). Software add-on vendors must also use standard formats for packaging and distributing add-ons. Furthermore, decentralized control over software evolution complicates product support and documentation since no single authority represents an application that has been evolved by multiple vendors. End-user installation of software add-ons necessitates that configuration management and analysis be robust and accessible to non-technical users. We focus on this class of evolution in the remainder of the paper.

3 EXISTING TECHNIQUES FOR SUPPORTING DECENTRALIZED SOFTWARE EVOLUTION

The degree of flexibility afforded by different approaches to DSE is depicted in Figure 1. At the extreme left of the spectrum, the software vendor deploys the application's source code, enabling anyone with sufficient expertise to modify any aspect of the application's functionality. Although this is rare in the commercial software market, numerous free-ware[1] applications adopt this approach, including the Linux OS, the Apache Web server, and the GNU tools (e.g., gcc, emacs). Netscape's Communicator is among the first commercial products to distribute source code for independent extension. Although the types of changes that can be made are unrestricted, it is extremely difficult to combine independently developed add-ons. This is because determining whether or not two changes conflict requires careful analysis of the source code and cannot be automated. The problem is analogous to merging several branches of a software revision tree in a software configuration

1. For a definition of "free software", see [6]

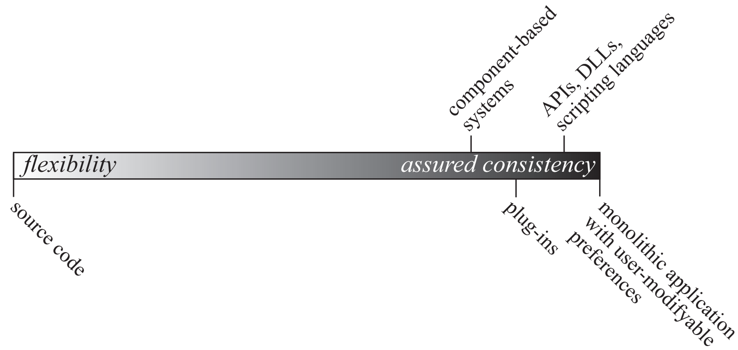

Figure 1. The trade-off between flexibility (on the left) and application consistency (on the right) made by different decentralized software evolution mechanisms

management system.

At the extreme right of the spectrum, the software vendor deploys the application as a monolithic entity, with a fixed set of user-selection options. A large majority of commercial software applications adopt this approach because (1) application consistency may be assured since a single software vendor has exclusive control over its evolution, and (2) a software vendor can protect their intellectual property rights by making it extraordinarily difficult for others to reverse engineer the application.

Applications in between these two extremes support some form of software evolution by trading-off assured consistency (right-end) for flexibility (left-end). These systems enable end-users or third-party software vendors to customize or extend the application's functionality *independent* of the original application vendor. A relatively small but growing number of software applications lay between these two extremes. Some examples include Microsoft's Office Suite, Adobe Photoshop, and Qualcomm's Eudora. The most common techniques for supporting DSE are briefly described below.

Application Programming Interfaces (APIs): An API is a set of functions that an application provides for other applications. APIs are commonly used as tool integration mechanisms since they enable other applications to invoke the services of the host application without user involvement. APIs provide a limited subset of the operations necessary to support evolution. For example, API-based software add-ons cannot replace or remove existing functionality, or interpose new functionality between existing parts. As a consequence, the host application's predetermined API circumscribes the class of possible changes.

Software plug-ins: The plug-in mechanism provides a place holder for third-party components. The host application specifies the requirements and functional interface that all plug-ins must adhere to, and provides a mechanism by which new plug-ins register themselves with the host. Netscape's Communicator, for example, registers plug-ins by placing them in a special file system directory queried on startup. The host application selects among the plug-ins and invokes them as necessary. Plug-in based software add-ons can only provide alternative implementations for behaviors anticipated by the original developers. The interposition and removal of functionality is not supported since dependency information between plug-ins cannot be determined.

Scripting languages & macros: A scripting language provides a domain-specific language for specifying behavior using language primitives and library functions. Scripting language-based mechanisms provide essentially the same flexibility as the API mechanism, except that the scripting language provides domain-specific language constructs that can facilitate the implementation of add-ons, and a built-in compiler and interpreter that lower the entry barrier for developing add-ons.

Dynamic link libraries (DLLs): Dynamic link libraries provide a late-binding mechanism whereby an application can load and link to an external software module during runtime. Applications employ DLLs to reduce runtime memory use and to share common functionality. A software add-on can augment, replace, or remove functionality by masquerading as an application DLL (e.g., by replacing the file representing the DLL in the file system). Balzer's instrumented connector technology [1] use this technique to alter Netscape's Communicator browser to support browsing of virtual, encrypted file systems. Although DLL-based software add-ons are unique in that they can be used to evolve an application in a manner unanticipated by its developers, they have two limitations. One, DLL add-ons can only be use in place of existing DLLs, which circumscribes the class of changes. Two, unexpected side-effects may result if the add-on DLL violates an unstated assumption between the application and the DLL.

Component-based applications: Component-based applications built using a component-object model, such as COM [3] or CORBA [11], are applications composed of separately compiled modules, called components, that are linked to one another during runtime. Since each application component exposes its interface, component-based applications expose a richer, more structured API, increasing the potential of supporting unanticipated changes. But since existing technologies do not try to separate application functionality from component communication, components tend to be riddled with hard-coded references to other components. This makes component replacement, removal, and interposition of behavior difficult.

All of the these techniques, except for source code, generally preserve only a small portion of the design's flexibility in the deployed system. Not only does this restrict the set of potential changes, but it precludes changes unanticipated by the original developers. Composition of software add-ons is also poorly supported by existing techniques. Most existing techniques circumvent the composition problem by preventing interaction between add-ons. This is indeed the approach advocated by Szyperski [18].

4 OUR APPROACH

Our approach to decentralized, post-deployment software evolution overcomes many of the limitations exhibited by previous approaches. Our approach is based on evolving applications at the software architectural-level [15,16]. Our approach to DSE is unique in following ways:

- We augment the abstractions provided at the architectural level with stylist rules that further facilitate DSE. We require all components to communicate through connectors, which are preserved as explicit entities in the implementation and act as communication routers. Furthermore, connectors encapsulate and localize the binding decisions between components, which makes it possible to change binding decisions without altering the components.
- We include the application's architectural model and a mechanism to maintain the consistency between this model and the implementation with the deployed system. The deployed architectural model describes the interconnections between components and connectors, and their mappings to implementation modules. The mapping enables changes specified in terms of the architectural model to effect corresponding changes in the implementation.
- We deploy an architecture evolution manager (AEM) with the application. The AEM maintains the correspondence between the architectural model and the implementation as changes are made. If a change violates application consistency, the AEM can reject the change.

Our approach enables software add-ons to leverage the architectural model's rich semantics to guide changes. This avoids many of the accidental difficulties incurred by existing approaches. Since the application's entire architecture is exposed and remains malleable in the deployed system, the design's plasticity is preserved and made accessibly to third-party add-on vendors. This overcomes the limited scope of change exhibited by existing approaches. Software add-ons can inspect and modify the system's architectural model in order to achieve the necessary semantics. This, for example, greatly simplifies the problem of interposing a new component between two existing components since the architectural model can be queried to locate the affected components and connectors. As independently developed software add-ons are installed and removed, the architectural model can be analyzed to detect inconsistencies. The availability of the architectural model can also be used to detect conflicts between independently developed add-ons.

5 RESULTS TO DATE

We have implemented a prototype tool suite, called ArchStudio, that implements our approach for applications implemented in the C2 architectural style [19]. ArchStudio's conceptual architecture is depicted and briefly described in Figure 2. We have used ArchStudio to implement two applications with several add-ons each. More details regarding ArchStudio and a sample application implemented using it are described in [12].

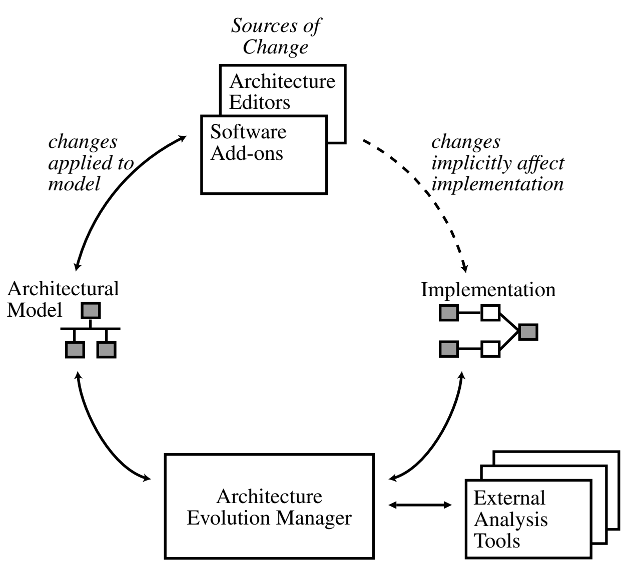

Figure 2. A conceptual architecture of the ArchStudio tool suite. *Software add-ons* evolve an application by inspecting and changing its *architectural model*. Changes may include the addition, removal, or replacement of components and connectors, or changes to the configuration of those components and connectors. The *Architecture Evolution Manager* is notified of changes and has the opportunity to revoke changes that violate system integrity. The Architecture Evolution Manager may utilize *external analysis tools* to determine if changes are acceptable.

The lower portion of ArchStudio is deployed with each application. When a user downloads a new software add-on using their Web browser, the add-on's installation script is located and executed.

We have implemented a simple end-user tool for installing and removing software add-ons, called the Extension Wizard, that is also deployed with the application. End-users use a Web browser to display a list of downloadable software add-ons, e.g., provided by the software vendor on their Web site. When the user selects the file representing the add-on, the Web browser downloads the file and invokes the Extension Wizard. The software add-on file is a compressed archive containing new implementation modules and an installation script. The Extension Wizard uncompresses the file, locates the installation script it contains, and executes it. The software add-on's installation script may query and modify the architectural model as necessary. The installation script an exception if the AEM detects a violation of application invariants. If the installation succeeds, the Extension Wizard notifies the end-user and provides an option to un-install the add-on.

6 CONCLUSIONS

Our results to date are encouraging, but several difficult issues remain. A general framework for ensuring application consistency is needed. Our current architecture evolution manager only enforces C2-style rules, which do not, by themselves, guarantee that changes will leave the application in a consistent state. Some aspects of the style do facilitate this type of in-the-field analysis. We are currently investigating the suitability of graph grammars, architectural constraints, and event-based resource models for representing application invariants. While these techniques hope to address software compositionality, new techniques are needed to address the problems of "composing" documentation and product support in a decentralized environment.[2]

7 REFERENCES

1. R. Balzer. Instrumenting, monitoring, & debugging software architectures. http://www.isi.edu/software-sciences/papers/instrumenting-software-architectures.doc, January 28, 1998.
2. G. Booch. *Object-oriented analysis and design*. Second edition. Benjamin/Cummings Publishing Company, Inc. 1994.
3. K. Brockschmidt. *Inside OLE 2*. Microsoft Press, 1994.
4. M. R. Cutkosky, J. M. Tenenbaum, J. Glicksman. Madefast: Collaborative engineering over the Internet. *Communications of the ACM*, vol. 39, no. 9, September 1996.
5. R. Fielding, G. Kaiser. The Apache HTTP server project. *IEEE Internet Computing*, July-August 1997.
6. Free Software Foundation. Categories of free and non-free software. http://www.gnu.org/philosophy/categories.html, January 28, 1998.
7. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
8. R. S. Hall, D. Heimbigner, A. van der Hoek, A. L. Wolf. An architecture for post-development configuration management in a wide-area network. *17th International Conference on Distributed Computing Systems*, Baltimore, Maryland, May 1997.
9. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. Aspect-oriented programming. *PARC Technical Report*, SPL97-008 P9710042. February 1997.
10. K. J. Lieberherr. *Adaptive object-oriented software–the Demeter method*. PWS Publishing Company. 1996.
11. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, July 1996. http://www.omg.org/corba/corbiiop.htm
12. P. Oreizy, N. Medvidovic, R. N. Taylor. Architecture-based runtime software evolution. To appear *International Conference on Software Engineering 1998,* Kyoto, Japan. April 1998.
13. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of ACM.* vol. 15, no. 12, December 1972.
14. D. E. Perry, G. E. Kaiser. Models of software development environments. *IEEE Transactions on Software Engineering*, vol 17, no. 3. pp 283-295, March 1991.
15. D. E. Perry, A. L. Wolf, Foundations for the study of software architecture. *Software Engineering Notes*, vol 17, no 4, October 1992.
16. M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
17. K. Sullivan, D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology.* vol 1, no 3, July 1992.
18. C. Szyperski. Independently extensible systems–software engineering potential and challenges. *Proceedings of the 19th Australasian Computer Science Conference*, Melbourne, Australia, January 31- February 2, 1996.
19. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, D. L. Dubrow. A Component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering,* June 1996.
20. Tivoli Systems Inc. *Applications Management Specification*. http://www.tivoli.com/
21. N. Wirth. Program development by stepwise refinement. *Communications of the ACM*. vol. 14, no. 4, April 1971.

2. This material is based on work sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government. Approved for Public Release - Distribution Unlimited.