Coping with Application Inconsistency in Decentralized Software Evolution

Peyman Oreizy and Richard N. Taylor

Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
+1 949 824 8438
{peymano, taylor}@ics.uci.edu

1 Introduction

Decentralized software evolution enables third-parties to change a software application independently of its original software development organization [10]. Popular approaches to decentralized software evolution (hereafter abbreviated DSE) include application programming interfaces (APIs), software plug-ins, scripting languages, component architectures, event-based systems, and open-source development. Application vendors employ DSE as a means of attracting additional users—and, consequentially, increasing their market share—since it opens up the *possibility* that a third-party modified version of the application would satisfy the needs of end-users unsatisfied with the original version. This benefits everyone involved: the original application vendor sells more product since customization implies use; third-party developers deliver a product in less time and at a lower cost by reusing software as opposed to building-from-scratch; and end-users receive a higher quality product, customized to suit their needs, in less time and at a lower cost as compared to building-from-scratch.

Numerous commercial and free-ware applications routinely demonstrate the benefits of DSE. Applications such as GNU Emacs, Adobe Photoshop, Microsoft Office, the Linux operating system, and the Apache web server, each offer a rich assortment of *add-ons,* developed by independent third-parties, that augment the application's functionality. For example, third-parties have used Adobe Photoshop's plug-in mechanism to add novel imaging effects and transformations, support new file formats, and support new hardware devices. Without DSE, end-users would have to rely on Adobe or a competitor to provide such features. But since some add-ons cater to niche markets, it's unlikely that Adobe would ever provide the breadth of functionality offered by third-parties.

In adopting a DSE mechanism, software producers struggle with a number of issues. One of these is *application consistency*[1], i.e., how can the software producer preserve the integrity of the application if third-parties can independently modify it? This problem is a consequence of the truism that "independent change introduces the possibility of inconsistent change."

This paper briefly explores application consistency in the context of decentralized software evolution. We begin in section 2 by surveying common strategies for dealing with application consistency in DSE. Section 3 proposes a new strategy based on resource models. Section 4 considers related work on non-DSE techniques for assuring consistency that could potentially be adapted to DSE.

2 Common Strategies in DSE

Application inconsistency in DSE can arise in two ways: (1) interactions between the host application and a single add-on, and (2) interference between two or more add-ons. Anomalies of the first type involve two self-aware parties (the third-party developer creates an add-on for a particular software producer's host application), whereas anomalies of the second type involve independent entities. This

1. Application consistency with respect to what? For the purpose of this paper, we use it generically to refer to any software quality that the software producer is concerned with, such as safety, timeliness, lack of deadlocks, etc.

difference has lead to different techniques for dealing with each source of inconsistency. We explore each one in the following subsections.

2.1 Host application/Add-on (In)Consistency

Host applications that use DSE offer a variety of resources to aid third-party developers in creating add-ons. Commonly provided resources include technical documentation covering pertinent portions of the host application's design and functional behavior, cookbook code and sample add-ons, discussion groups and mailing lists, and test scripts. Such resources not only help third-parties implement add-ons, but help them to implement add-ons "correctly", i.e., in a way that preserves host application consistency. But using these resources does not *ensure* application consistency. Software producers rely solely on the good intentions and thoroughness of third-party developers in developing and testing their add-ons using traditional software testing and analysis techniques. Even so, these are the most commonly used techniques for avoiding application inconsistency.

If the intentions of third-parties are in doubt, there are two general strategies for preventing inconsistency: (a) restricting the language used to implement add-ons, and (b) restricting the runtime environment of the add-on. We consider each of these in turn below.

Restricting the language for implementing add-ons amounts to circumscribing the range of operators and operands available to add-ons. This avoids certain classes of inconsistencies all together by eliminating language constructs that lead to those errors. For example, the Java programming language does not provide pointer types, which prevents potential misuses of direct memory access and pointer arithmetic. Examples of more restricted add-on implementation languages include spreadsheets formula languages and database query languages such as SQL.

But restricted add-on implementation languages are not a panacea. If the language is Turing-complete, no amount of automated analysis can prove that the add-on terminates (due to the halting problem). As a consequence, only certain aspects of consistency may be guaranteed. The software producer must still rely in part on the good intentions of third-party developers.

Restricting the add-on runtime environment is achieved in one of two ways: program interpretation and/or software fault isolation. Program interpretation prevents certain types of inconsistencies by verifying program instructions before executing them. Examples of this technique include Emacs' elisp and Java's virtual machine interpreter. Although interpretation usually occurs at the level of the source language or on an intermediate executable format such as bytecodes or p-code, modern virus checkers interpret native machine instructions to detect and prevent malicious operations by software viruses [9].

Software fault isolation techniques, such as sandboxing [13][16], achieve similar results with non-interpreted languages. In sandboxing, critical points of the compiled program are "instrumented" with logic that verifies particular constraints on the program's runtime environment. MisFIT [12], a software fault isolation tool, instruments assembly language programs by inserting code before every instruction that references memory to verify that only permissible memory locations are addressed. Other similar mechanisms include the invalid instruction interrupts of modern hardware processors, the memory and file system protection mechanisms of operating systems, and Java's virtual machine security manager.

2.2 Multiple Add-on (In)Consistency

Although it is reasonable to expect a third-party developer to thoroughly test their add-on with the host application before releasing it to the public, such testing cannot reveal inconsistencies that may arise when it is used in conjunction with other third-party add-ons. Given a host application with n add-ons, there are 2^n unique host/add-on configurations (including the degenerate case of the host application without add-ons). Thus, an application with 20 add-ons has over one million unique configurations. Obviously, this makes exhaustive testing of configurations infeasible.

Software producers use a variety of strategies to combat the problem. We have generalized them into three categories: defensive design, standardized interfaces, and external tools. We examine each of these in turn using several examples from real-world applications.

A *defensive design* strategy avoids inconsistencies altogether by having the host application developers design and implement the host application in such a way that add-ons are forced to be mutually independent—they do not depend on or interact with one another directly. Instead, the host application mediates all interactions among add-ons. Szyperski [15] refers to such system as *independently extensible* as they "can cope with the late addition of extensions without requiring a global integrity check." Krishnamurthi and Felleisen [5] have formalized a more restricted notion of independent extensibility. Many operating systems and shrink-wrapped applications adopt this strategy. For example, Netscape's Communicator Web browser supports a build-in set of audio and graphical media types, and allows third-parties to incorporate new media types using a plug-in mechanism. Each plug-in implements a standard interface and associates itself with particular media types when Communicator begins executing. Communicator invokes the plug-in whenever a Web page containing that content type is retrieved. If a single Web page contains several different content types, each plug-in is provided with an independent means of retrieving that content and its own private region of the display screen to render the content. In this way, two plug-ins will not interfere with one another if they behave as directed by the host application. Although a defensive design strategy can be used to prevent most types of inconsistency, it has its drawbacks. The host application must mediate all the interaction between add-ons, which precludes the use of novel and useful interactions that were not foreseen by the software producer. Additionally, consistency is preserved so long as the plug-ins do not inadvertently interact through some shared operating system resource, such as a device or file.

A *standardized interface* strategy, used by component-based architectures built on OMG's CORBA, Microsoft's COM, and FIELD [11], assumes that interface compatibility implies behavioral compatibility. For example, FIELD's debugger understands 34 command messages and broadcasts 22 event messages, each of which has its own ASCII string representation. Other tools in the environment (i.e., add-ons) can invoke debugger commands by sending it appropriately formatted messages. As a result, independently developed add-ons can interact with one another if their interfaces are compatible. But interface compatibility is a poor measure of consistency since it does not prevent two (or more) add-ons from using the same component interface in incompatible ways. In the debugger example, two add-ons that simultaneously send a command to the debugger to 'single step' will end up advancing the program by two instructions instead of the intended single instruction.

An *external tool-based* strategy uses tools to detect and resolve inconsistencies. The "system extension" mechanism of Apple Computer's System 7 operating system uses such a strategy. In this case, system inconsistencies can arise when two or more add-ons, called *extensions* or *INITs* on the Macintosh, intercept invocations to the same operating system function (i.e., "patch a trap") in incompatible ways. For example:

> "[O]ne INIT may assume that a particular trap works in a particular way and only returns one of two possible error codes. If another INIT comes along and also patches that trap, it may decide that it needs to force the original to return yet a third error code. Depending on the order in which the INITs load, which decides the order in which the patches themselves are called, one may not expect the new error code and your code may unexpectedly crash or behave strangely. Issues like this, including many that are much more subtle, are the cause for many problems between system extensions."
>
> — Zobkiw [17], page 253

Since there is no way to determine if two or more add-ons will work together without trying them together, there is no built-in mechanism for detecting and resolving inconsistencies. Instead, system integrators and end-users use utility programs to help detect and resolve inconsistencies in a semi-automated way. For example, Casaday & Greene's Conflict Catcher utility attempts to resolve inconsistencies by repeatedly rebooting the machine with different combinations of system extensions until it narrows the list of potential culprit extensions down to one. Such tools rely on the end-user to determine if the anomaly persists after each reboot. This can be an arduous, time-consuming, and an error-prone process. The Complete Conflict Compendium [www.mac-conflicts.com] and MacFixIt [www.macfixit.com/archive.html] are examples of related strategies. These Web sites simply catalog end-user reported conflicts between Macintosh system extensions. End-users manually search these catalogues to find remedies to extension conflicts.

3 A Resource-based Strategy?

Operating systems suggest an alternate strategy. Operating systems effectively prevent multiple, independently developed applications from interfering with one another even though the applications must share common functional interfaces and resources such as the system services, processor, memory, file system, and hardware devices. Operating systems essentially do this by managing the resources accessible to applications. For example, the speaker device is usually reserved for a single application. Applications must request exclusive access to the device before using it; attempts to use the device without permission are prevented. More complex resource sharing policies are used for the processor, memory, and file system.

An operating system-like resource strategy may be used in the debugger example by modeling the 'single step' command as an exclusive access resource. Add-ons would then be required to "request permission" to send the 'single step' command from a resource manager. The resource manager could then ensure that a single add-on has permission to send the 'single step' command, and prevent other add-ons from using it without permission.

Various twists on the notion of a resource may be used to prevent a broad class of add-on inconsistencies. For example:

- Exclusive access resources, which allow only one add-on at a time to own the resource, can be used to prevent two or more add-ons from using the debugger's 'single step' command, or patching the same Macintosh system trap, or responding to the same GUI event, or handling the same Netscape content type.
- Bounded access resources, which allow at most *n* add-ons to own a resource, can be used to reserve bandwidth on a critical communication channel.
- Spacial resources, which allow at most one add-on to own a region of the space, can be used to prevent add-ons from writing to overlapping regions of a graphical user-interface.
- Temporal resources, which allow multiplexed access to a resource by several add-ons, can be used to guarantee several add-ons equal opportunity to a shared communications channel.

This strategy is also not without its drawbacks. The technique relies on the software producer to predict all the types of inconsistencies that may occur and to devise resources models that can be used to prevent them. Deadlocks may occur as add-ons wait for mutually unavailable resources.

4 Related Work

Assuring consistency in any large application is difficult. With DSE, the problem is aggravated by (1) independent change and (2) incomplete access to the implementation by third-parties.

A number of techniques for detecting inconsistency in traditional centralized software evolution contexts could potentially benefit DSE. These include, for example, reuse contracts [7][14],

configuration graphs [3], grammars [1], architectural type theory [6], and architectural constraints [8]. Reuse contracts explicitly capture contractual obligations between the components of a system and evolutions of those components. Hence, when two independent changes are merged, the contracts can be consulted to determine if the changes are incompatible. Configuration graphs explicitly capture two forms of dependencies between application modules: (1) if x is in the system, so must y, and (2) if x is in the system, y cannot be in the system. These dependencies are verified whenever changes are made to determine if the system is in a consistent state. Batory and Geraci use grammars and a design rule checking system based on pre- and post- conditions to describe valid system compositions. Medvidovic combines type theoretic notions of compatibility and pre- and post-conditions to determine valid configurations of software architectures. Architectural constraints are a more general form of configuration graphs and can express constraints over the topology of a set of interconnected components. These constraints could be verified at a variety or mixture of times, including build time, when add-ons are added or removed from the application, or during runtime.

Techniques based on source code analysis are generally inadequate. This is because determining whether or not two arbitrary source code changes conflict requires careful analysis of the source code, and typically cannot be automated. The problem is analogous to merging several branches of a software revision tree in a software configuration management system (see [2][4]).

5 References

[1] Don Batory, Bart J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, vol. 23, no. 2, February 1997.

[2] J. Buffenbarger. Syntactic Software Merging. *Proceedings of the Fifth Workshop on Software Configuration Management*. 1995. Pages 153-172.

[3] Matti A. Hiltunen. Configuration Management for Highly-Customizable Services. *Proceedings of the Fourth Internation Conference on Configurable Distributed Systems*. May 1998.

[4] S. Horwitz, J. Prins, T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*. vol. 11, no. 3, July 1989, Pages 345-388.

[5] Shriram Krishnamurthi and Matthias Felleisen. Toward a Formal Theory of Extensible Software. *Proceedings of the Sixth Symposium on the Foundations of Software Engineering*, 1998.

[6] Nenad Medvidovic, Richard N. Taylor, and David S. Rosenblum. An architecture-based approach to software evolution. *Proceedings of the First International Workshop on the Principles of Software Evolution*, Kyoto, Japan, April 20-21, 1998. Pages 11-15.

[7] Tim Mens. A Basic Formalism for Systematic Software Evolution. *Proceedings of the First International Workshop on the Principles of Software Evolution*. Kyoto, Japan, April 20-21, 1998.

[8] Robert T. Monrow. Capturing Software Architecture Design Expertise with Armani: The Armani Language Reference Manual version 1.0. *CMU Technical Report CMU-CS-98-163*, School of Computer Science, Carnegie Mellon University, October 1998.

[9] Carey Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, vol. 40, no. 1, Jan. 1997. Pages 46-51.

[10] Peyman Oreizy. Decentralized software evolution. *UCI-ICS Technical Report 98-42*. December 1998.

[11] Steven P. Reiss. *The FIELD programming environment: A friendly integrated environment for learning and development*. Kluwer Academic Publishers. 1996.

[12] Christopher Small. A tool for constructing safe extensible C++ systems. *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. Portland, OR, USA, 16-20 June 1997. Berkeley, CA, USA. 1997. p. 175-84.

[13] Christopher Small and Margo Seltzer. A Comparison of OS Extension Technologies. *Proceedings of the USENIX Technical Conference 1996*, New Orleans, LA, p41-54, January 1996.

[14] Patrick Steyaert, Carine Lucas, Kim Mens and Theo D' Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. Proceedings of OOPSLA 1996. Also published in ACM SIGPLAN Notices, 31(10), pp.268-286. ACM Press, 1996.

[15] Clements Szyperski. Independently extensible systems–software engineering potential and challenges. Proceedings of the *19th Australasian Computer Science Conference*, Melbourne, Australia, January 31- February 2, 1996.

[16] R. Wahbe, S. Lucco, T. Anderson, S. Graham. Efficient Software-based Fault Isolation. *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. Asheville, NC. p203-216, December 1993.

[17] Joe Zobkiw. *A Fragment of Your Imagination: Code Fragments and Code Resources for Power Macintosh and Macintosh*. Addison-Wesley Pub. Co., August 1995.