On the Differences Between Very Large Scale Reuse and Large Scale Reuse

Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

dsb@cs.utexas.edu

Abstract

Very large scale software reuse (VLSR) is an every-day occurrence whereas large scale software reuse (LSR) remains an elusive and unrealized goal. We offer an explanation for this apparent contradiction in this paper.
Keywords: Genesis, realms, software building-blocks.

1 Introduction

Very large scale software are systems of 50K lines of code or greater. As-is reuse is the reuse of software without modifications. As-is very large scale software reuse (henceforth called *as-is VLSR*) is an every-day phenomena. Each time we build an application on top of Unix, we are reusing Unix as-is. Each time we build a graphical front-end using Motif or Interviews, we are reusing Motif or Interviews as-is. Since both Motif and Interviews are built upon X-windows, X-windows is reused as-is. Creating pipes in Unix, where processes of prewritten file filters are glued together, is yet another example. In general, as-is VLSR occurs anytime we build applications upon existing software.

There are two general conditions for as-is reuse to occur. They are when users (1) agree to use the abstractions provided by existing software and (2) agree to use the existing implementations of these abstractions. Both points are self-evident: if the abstractions presented by software components in a library are not relevant to an application, they obviously will never be (re)used. Similarly, if performance or special features are critical requirements and existing software doesn't provide either, then that software cannot be (re)used without modification.

There probably is universal agreement that in the ideal case software components should be reused as-is. Since as-is VLSR is already is a daily occurrence, and large scale reuse (LSR) is known to be a fundamentally difficult problem, where is the contradition? Why has as-is VLSR been successful while as-is LSR has not? To answer this question, we introduce a model of VLSR and LSR.

2 A Model of As-Is Reuse

From our work on Genesis and Avoca [Bat88, Bat91a, Bat91b, OMa90, Pet90], we have shown that large software systems correspond to type expressions. Our idea is simple. Let R be the function-call interface to a software system. In principle, we know that there can be many different implementations of R. We define the *realm* of R to be the set of all software systems that support exactly interface of R. The members of realm/interface R can be listed as an enumerated type. Two possible realms R and S are shown below:

R = a, b, c

S = d[x:R], e[x:R], f[x:R]

Interface R has three implementations, namely the software components a, b, and c. Similarly, interface S is implemented by the components d, e, and f. Observe that realms group together all components that are *plug-compatible* and *interchangeable*. For example, because all components of R implement the *same* interface, one can swap one implementation/component of R with another. The same applies for components of other realms.

Components may have parameters. All components of S, for example, have a single parameter of type R. What this means is that a component d[x:R] of S translates objects and operations of the interface of S to objects and operations of the interface of R. The translation itself *does not depend* on how R is implemented. This means that *any* implementation of R can be 'plugged' into component d to make it work.

Software systems correspond to *type expressions*. Two systems are shown below:

System_1 = d[b]

System_2 = f[b]

System_1 is a composition of component d with b; System_2 is a composition of component f with b. Since both of these systems present the same interface (i.e., both present the interface of S), System_1 and System_2 are also interchangeable implementations of S.

The way we have modeled software makes as-is reuse easy to spot. Consider two systems and their type expressions. If both expressions reference the same component, then that component is being reused. Note that System_1 and System_2 reuse component b. More generally, if two systems have a common subexpression, then these systems share a common subsystem.

Finally, observe that our model of software construction is independent of component size. Components can be very large (i.e., the size of Unix and X-windows), or they could be substantially smaller [Bat91a, Bat91b, Bat91c]. Thus, our model can be used to explain as-is VLSR as well as as-is LSR.

3 An Example

Consider the following realms. Let UNIX denote the realm of systems that implement the Unix interface. Let XW denote the realm of X-window interface implementations, and MOT be the realm of implementations of the Motif interface. Finally, let APPL be the realm of applications whose interface takes command-line and mouse-click inputs. Typical population of these realms are:

UNIX = bsd4.3, system5

XW = xwindows[x:UNIX]

MOT = motif1.1[x:XW]

APPL = myprog[x:MOT, y:UNIX], ...

We have taken a few liberties in defining the above realms to keep our example simple. Note that we have lumped bsd4.3 and system5 together in the same realm (UNIX). Actually, there are significant differences between Berkeley Unix and System 5, and in reality each belongs to its own distinct realm. For our purposes, however, both implement a common subset of interface functions which we define as the UNIX realm interface. Another point worth mentioning is that we have simplified our model of the motif[ ] and xwindows[ ] components by omitting parameters (i.e., their calls to the Unix interface). We leave it to readers as a straightforward exercise to eliminate these simplifications.

Recall that systems correspond to type expressions. Suppose 'myprog' is an application program that makes calls to the Unix and Motif interfaces. Motif, in turn, translates its calls into X-window calls, which in turn, translates into calls to Berkeley Unix. The layering of very large scale software that defines this system is given by the expression:

My_System = myprog[motif1.1[xwindows[unix4.3]], unix4.3]

By swapping the unix4.3 component with system5, we have ported My_System to System 5 Unix.

As a step toward understanding the problems of LSR, consider the membership of contemporary realms. As a general rule, most realms today have very few components. The members of the UNIX realm, for example, are the current implementation(s) of Unix plus out-dated versions. The same for XW (X-windows) and MOT (motif). The reason is simple: interfaces to software systems tend to be ad hoc and unique. The institution that designed the interface of a system also built that system, and maintains only one implementation of it (modulo version and platform upgrades). The idea of building a family of different implementations for an interface is not yet popular. Consequently, typical realms have singleton (or very small) memberships. This provides us with a key clue to resolving the contradiction we identified earlier.

4 Explaining the Contradiction

As mentioned earlier, as-is VLSR is an every-day occurrence while as-is LSR remains an unrealized goal. The techniques that have been proposed for as-is LSR (e.g., formal methods) have had difficulties scaling-up. Oddly enough, the simple ideas that have made as-is VLSR work have not been successfully *scaled-down* so that large (sub)systems like unix4.3, xwindows, motif1.1, etc., can themselves be defined from compositions of more primitive components.

Recall the two conditions for successful as-is reuse. Users must (1) agree to use the interface abstractions of existing software and (2) agree to use the implementations of these interfaces. As a community, we have not been successful in agreeing on the interfaces/abstractions that lie inbetween the interfaces of major systems (e.g., interfaces/abstractions in between Motif and X, between X and Unix, etc.). Nor have we, as a community, truely realized the importance of families of systems that share the same interface but have different implementations.

Until these two issues are bridged, as-is LSR will continue to remain elusive and unrealized. In [Bat91a], we explore these issues further, and present a validated approach for achieving as-is LSR.

References

[Bat88] D.S. Batory, 'Concepts for a Database System Synthesizer', ACM PODS 1988, 184-192.

[Bat91a] D.S. Batory and S.W. O'Malley, 'The Design and Implementation of Hierarchical Software Systems Using Reusable Components', submitted for publication.

[Bat91b] D.S. Batory and S.W. O'Malley, 'Genvoca: Reuse In Layered Domains', Proc. 1st International Workshop on Software Reuse, Dortmund, Germany, 1991.

[Bat91c] D.S. Batory and S.W. O'Malley, 'A Definition of Open Architecture Systems with Reusable Components: Preliminary Draft', ICSE Domain Modeling Workshop, 1991.

[Pet90] L. Peterson, N. Hutchinson, and H. Rao, and S.W. O'Malley, 'The x-kernel: A Platform for Accessing Internet Resources'. IEEE Computer (Special Issue on Operating Systems), 23,5 (May 1990), 23-33.

[OMa90] S.W. O'Malley and L. Peterson, 'A New Methodology for Designing Network Software', University of Arizona TR 90-29 (Sept. 1990). Submitted for publication.

5 Biography

Don Batory is an Associate Professor in the Department of Computer Sciences at The University of Texas, Austin. He received his Ph.D. from the University of Toronto in 1980, he was Associate Editor of the IEEE Database Engineering Newsletter from 1981-84 and was Associate Editor of ACM Transactions on Database Systems from 1986-1991. He is currently a member of the ACM Software Systems Award Committee, and his research interests are in extensible and object-oriented database management systems and large scale reuse.