

Software Malleability: We're Losing it!

Jeffrey Voas (jmvoas@rstcorp.com)
Reliable Software Technologies
21351 Ridgetop Circle, Suite 400
Sterling VA 20166 USA
703-404-9293

Abstract

There are three fundamental reasons why systems are written in software versus being built in hardware: (1) software does not wear out or decay, (2) software can be mass distributed at low replication costs, and (3) software can be changed quickly if defects are found or new features are needed. In contrast, hardware does decay, has greater manufacturing costs per unit, and requires that each system be individually modified. For example, releasing a new software version or sending out a patch is trivial when compared to recalling hardware or shipping modified hardware.

Although each advantage provides enough breadth and depth for an entire article, this position paper focuses on the third advantage. In particular, we will focus on how Commercial-Off-the-Shelf (COTS) software decreases the third advantage from all who license it. Why? Because COTS software is delivered to users and system integrators in executable format. This “check-mates” those parties from modifying it. They are stuck with precisely what was delivered. Because of this, their ability to maintain systems that rely on COTS software is reduced. And this absence of knowledge concerning what the COTS software can and cannot do also increases their exposure to damage if the COTS software is defective or malicious.

1 Introduction

The adage, “if you want something done right, do it yourself” is less of an option for software developers today than it was 10 years ago. Today’s software systems are “systems of systems of systems”, and developers must accept the fact that substantial portions of these composite systems will have been written by unknown developers.

The loss of control over *every* aspect of a system’s functionality is worrisome to those persons responsible for quality at the system-level. Those parties need assurances that the acquired software components can tolerate each other. But if the acquired software is delivered in a executable format that is nothing more than a “black box”, the question becomes “how?”

Software reuse has the potential to massively increase the rate at which information systems are built while simultaneously reducing development costs. Software reuse occurs

in different ways, including: (1) purchasing Commercial-Off-The-Shelf (COTS) “generic” software, and (2) reusing one’s own software components from project to project by keeping them in reusable, internal code libraries. But each of these methods carry the risk that the complete system will suffer from problems associated with reused or acquired software.

Regardless, software reuse is easy to justify to project managers and corporate officials. Development costs can be significantly reduced given that the best programmers only churn out 10 lines of documented and tested code per day [8]. That is low productivity and a burden on time-to-market considerations. And it remains an easy sell to management as long as you do not raise the “red flag” that warns about potential downstream maintenance problems.

Even if COTS reuse does not engender maintenance problems, it is still not a panacea. The Ariane 5 disaster demonstrated that reusing good code can still lead to a disaster if the new environment is different [9].

Software reuse also allows us to field bigger and more complex software systems than we could afford if we had to build them from scratch. Thus, we are gaining access to products that we would never be able to have without massive quantities of reuse. For example, according to Reme Bourguignon, V.P. of Philips in Holland (as well as numerous public quotes made by Airbus officials), the amount of software embedded in a typical device is doubling in the number of lines about every 18 months. And much of that software is reused from version to version.

But let’s step aside for a moment and consider the average quality of commercial software. According to Les Hatton [6]:

“The industry standard for good commercial software is around 6 defects per KLOC in an overall range of around 6-30 defects per KLOC [12]. In addition, very few systems have ever stayed below one defect per KLOC ... even in safety critical systems.”

Surprisingly, this rate of 6-30 KLOC has held fairly constant for the last two decades, regardless of the shift to object-oriented technology, automated debuggers, better test tools and compilers, stronger type safety in languages (e.g., JAVA and ADA), etc. This suggests, then, that when we double the size of a system every 18 months, we are also doubling the number of defects.

What is also surprising is the rapid move toward acquiring more and more *application* Commercial-Off-The-Shelf (COTS) software in light of the well known and ubiquitous software quality problems. Consider that in 1997, 25.5% of a typical corporation’s information technology (IT) portfolio was COTS software. In 1998, that figure was expected to jump to 28.5%. And in 2002, the figure is expected to be 40% or greater [10]. The ultimate question then becomes:

If 40% of the software used by corporations has 6-30 defects per KLOC, and all of that code is in executable format, how will those corporations ever survive?

After all, the corporation’s ability to maintain the software components of those systems will be negligible.

This set of facts and anecdotes leads me to a position statement concerning the maintainability of COTS-dependent systems:

As we move toward licensing more and more commercial software, our ability to maintain the systems that rely on the COTS software will be substantially reduced.

Why? Because visibility inside of acquired, executable software is non-existent. In fact, about the only information that we get when we license such software is unfortunately supplied by the publisher. That information will mostly discuss the software's interfaces. That is not much visibility! And who knows if it is the "full truth."

My point here is that documentation about interfaces is woefully inadequate in order to be able to maintain these systems. To be able to perform system maintenance in a timely and correct manner, we need to be able to see "under the hood" of the acquired software. But as you know, we cannot.

2 Disposable Software?

Nonetheless, governments and commercial organizations are gearing up for systems that are heavily comprised of COTS software. Guidelines already exist from the US Federal Aviation Administration, US Department of Defense, National Research Council of Canada, US Food and Drug Agency, Electric Power Research Institute, and US Nuclear Regulatory Commission [5, 3, 4, 1, 2, 7, 11]. While these initiatives provide a good starting point, the real question that these standards must address is the downstream maintenance costs. These standards do not, chiefly because the "buy" versus "build" paradigm is so new that not enough historical data has been archived to allow for such considerations.

This brings me to a conclusion:

Systems may be deployable more quickly and cheaply using COTS software, but as time move on and upgrades become necessary, those upgrades may not be available or usable.

While disappointing, there may be a "silver lining" here. The trend toward more and more acquired software may move us toward the same maintenance paradigm that we have today with respect to personal computers:

Buy a new computer every two years instead of upgrading processors.

So the question then becomes whether we'd be willing to accept this? The answer is simple: we may have no choice. Personal computers are no longer upgradable and no one complains. They are not upgradable, in part, because they are too complex and manufacturers enjoy selling us a new computer every 2 years. To make this palatable, manufacturers keep prices reasonable and provide substantially improved computers on a regular basis.

The same lack of system-wide maintainability may soon move us toward thinking about *disposable* information systems as our first choice of maintenance paradigm. I admit that disposal is a radical approach, but if systems are truly unmaintainable as a result of COTS software, this may be the only option.

3 Maintenance Options and Conclusions

Before ending this article, I'd like to share my recommendations on how to deal with the problems of maintaining systems that rely on COTS software. Clearly we are not yet to the point where disposability is a plausible maintenance option since: (1) acquired software is not cheap enough yet, and (2) the competitive component marketplace necessary to provide alternatives is not here either. Therefore, although I feel that disposal is likely to be a plausible alternative someday, it is not yet today.

Note that the following recommendations are in no particular order. They simply represent what I consider to be plausible ideas that can be implemented today:

1. Make a reasonable attempt to convince yourself that you can build the code that you are considering acquiring yourself. If you do this and fail, that is okay. This is simply a due diligence step that you do not want to avoid,
2. Perform a business case analysis that assumes total and complete loss of maintainability at some time in the future. Assume worst case parameters such as: (1) the acquired software's source code being frozen, and (2) source code access is not attainable. Further, assume that there are no competitors from which to get an alternate product. Determine the impact of such a scenario on your business before settling for COTS [14].
3. Perform component regression testing and performing other activities (such as visiting a vendor's site) in order to pre-qualify components,
4. Develop procedures for determining when to accept upgrades,
5. Force the supplier to put their source code in escrow if they opt to discontinue the product, go out of business, or quit providing upgrades,
6. Decide ahead on how you will maintain source code if you get it from escrow,
7. Perform black-box, system-level testing after components are swapped in,
8. Perform software fault injection [13] at the interfaces to see how errors propagate across the acquired software and throughout the system, and
9. Develop a plan if the supplier is continually providing you with "customized" versions of their offering that are "one offs" from the versions that other licensees receive. One offs are really not generic COTS. They are specialized COTS. This increases your dependence on them and places you in a weakened negotiating position for future customizations.

In summary, we are building unmaintainable systems when we rely on COTS software. Whether the costs savings from employing COTS will remain high after the maintenance and supportability problems (that will arise later) are factored in is unclear. And whether we will someday have a competitive software component marketplace is also unclear given the trend for fewer and fewer companies as the result of mergers and acquisitions. If we do not have such a marketplace, it is unlikely that disposability will be a viable approach to maintenance activities.

References

- [1] US FOOD AND DRUG ADMINISTRATION. Reviewer Guidance for Computer Controlled Medical Devices Undergoing 510(k) Review, 1991.
- [2] US FOOD AND DRUG ADMINISTRATION. ODE Guidance for the Content of Premarket Submission for Medical Devices Containing Software, draft 1.3, dated 12 August 1996, available at: <http://www.fda.gov/cdrh/ode/dtswguid.html>.
- [3] DEFENSE SCIENCE BOARD. Acquiring Defense Software Commercially. June 1994.
- [4] NATIONAL RESEARCH COUNCIL OF CANADA. Project Summary, 1996.
- [5] FEDERAL AVIATION AUTHORITY. Software Considerations in Airborne Systems and Equipment Certification, 1992. Document No. RTCA/DO-178B, RTCA, Inc.
- [6] L. HATTON. Does OO Sync With How We Think? *IEEE Software*, 15(3), May 1998.
- [7] ELECTRIC POWER RESEARCH INSTITUTE. Guideline on Evaluation and Acceptance of Commercial Grade Digital Equipment for Nuclear Safety Applications (draft) 1996.
- [8] S. BAKER, G. McWILLIAMS, AND M. KRIPALANI. "Forget the Huddled Masses: Send Nerds", *Business Week*, July 11, 1997.
- [9] Prof. J. L. LIONS. Ariane 5 flight 501 failure: Report of the inquiry board. Paris, July 19, 1996, available at <http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html>.
- [10] META. *INforum* META Group, Inc., Stamford, CT. pp. 23, 1998.
- [11] NATIONAL RESEARCH COUNCIL. Digital Instrumentation and Control System in Nuclear Power Plants: Safety and Reliability Issues (Final Report), 1997. National Academy Press.
- [12] J. D. MUSA, A. IANNINO, AND K. OKUMOTO. *Software Reliability Measurement Prediction Application*. McGraw-Hill, 1987. ISBN 0-07-044093-X.
- [13] J. VOAS. Certifying Off-The-Shelf Software Components. *IEEE Computer*, 31(6):53–59, June 1998.
- [14] J. VOAS. COTS Software: The Economical Choice? *IEEE Software*, 15(2):16–19, March 1998.

About the Author

Jeffrey Voas is a Do-founder and Chief Scientist of Reliable Software Technologies. Voas has recently served as a Principle Investigator on efforts for NASA-Langley, NASA-Ames, DARPA, National Science Foundation, the USAF, and the US Army. He has published over 140 papers and co-authored two books: *Software Assessment: Reliability, Safety, Testability*

(John Wiley & Sons, 1995) and Software fault-injection: inoculating programs against errors (Wiley & Sons, 1998). Voas was the special editor of the IEEE Computer theme issue on “Commercial off-the-shelf software” (June 1998) as well as for the special issue of IEEE Software focusing on “Software Certification” (July 1999).

Voas was the General Chair for COMPASS'97 and serves on the Editorial Board for the Software Quality Professional Journal, IEEE Software and IT Pro magazines. Voas is on the Board of Trustees for the Center for National Software Studies, is the Executive Secretary of the IEEE Reliability Society, is the Program Chair for the 1999 International Symposium on Software Reliability Engineering (ISSRE), is Co-Program Chair for the International Conference on Software Maintenance '2000, and is the Program Chair for ECBS '2001. In 1994, the Journal of Systems and Software ranked Voas 6th among the 15 top scholars in Systems and Software Engineering. Voas was given the District of Columbia Council of Engineering and Architectural Societies's Young Engineer of the Year award in 1999. Voas is a senior member of IEEE and received a Ph.D. in computer science from the College of William & Mary in 1990. Voas's current research interests include information security metrics, software dependability metrics, software certification, software liability law, and information warfare tactics.