

# Upgrading Software Maintenance for Components

Jeffrey Voas

Reliable Software Technologies

## Abstract

Maintaining well-documented source-code is a fairly straight-forward process when it is performed by the code's original author. Because future information systems will contain functionality (provided in software units termed *components*) that was developed by organizations external to the organization responsible for assembling the system, traditional software maintenance processes may need to be modified or replaced. This paper explores what software maintenance technologies will be needed in order to successfully maintain component-based systems. We also explore the theory that maintenance will someday be easier to perform because new components can be "swapped-in" (as opposed to the usual source-code modification processes).

## Keywords

Software maintenance, components, Trojan horse, testing, software reuse, dependability.

## 1 Introduction

Software maintenance is the life-cycle phase that keeps an operational software system functioning even though the system's requirements are changing. Software maintenance occurs for one of two reasons: (1) to fix existing defects, or (2) to add new functionality to a system's existing functionality.

Software is, by its nature, highly malleable. Change as little as one line of code and you can have a very different system.

As we have learned from the Y2000 problem, malleability, while the key benefit of software over hardware, can come at a very high cost. It has been speculated that the total cost of "fixing" this problem globally is \$600B [2].

The Year 2000 experience has also taught us that a system's lifetime can exceed its original estimate by decades. It is not uncommon to see systems on the drawing board with life expectancies of 30+ years. The FAA's new air traffic control system, Wide Area Augmentation System (WAAS), has no plans for de-commissioning [4]. In theory, WAAS

will last forever. Such systems will spend almost all of their lifetimes in maintenance. We will need appropriate maintenance technologies unless we are willing to accept abbreviated life expectancies.

Systems that have existed for many years eventually receive the title of “legacy software.” Typically, legacy systems have historical records of satisfactory performance. Software systems with good reliability track records are hard to abandon when the alternative is new systems with unproven operational performance. Maintaining older systems is often less costly than developing new systems from scratch. These facts lead to systems surpassing their life expectancies.

Legacy software is not without problems, however. Legacy software often suffers from incompatibilities when combined with newer hardware and newer operating systems. Further, legacy logic can be difficult to interpret during maintenance because of hidden assumptions, long forgotten programming languages, and *dead* code.

Historically, code maintenance has been the responsibility of the organization that wrote the original software. The key reasons for having the development organization be tasked with the maintenance are:

1. The development organization knows the code better.
2. The dollar costs of maintenance should be lower than hiring a different organization to spin-up and learn the code.

But today software systems are becoming less tied to a single vendor and are instead the result of the efforts of a team of vendors. Boeing’s 777 aircraft is rumored to contain parts from every country in the world; so might future software systems.

If this occurs, software maintenance will become a team effort as well. But if any team member opts to not maintain their portion of the system or maintains their portion in a manner that is in conflict with the rest of the team, the team will encounter difficult maintenance problems. Because this “team approach” is the manner by which future systems will be built, the problem of maintaining such systems that are composed from multi-vendor components is the focus of this paper.

## 2 Systems Aren’t What They Used to Be

Today’s common software maintenance processes (*e.g.*, impact analysis, regression testing, *etc.*) originated when most systems were collections of subroutines and procedures that existed in source-code format.<sup>1</sup> For those systems, an impact analysis such as slicing [10] was often sufficient for seeing whether maintenance in one procedure could affect another

---

<sup>1</sup>*Impact analysis* is a collection of analyses that determine whether one part of a software system can affect another part, and if so, how. *Regression testing* is the process of testing code on inputs already used to test earlier versions to ensure that maintenance activities have not broken the software.

procedure. Impact analysis could then be used to bound the amount of re-testing necessary to only the affected procedures.

But the procedural paradigm for software development is being replaced by object-oriented design. Object-oriented design has drastically changed how systems are built and object-oriented design (along with “glue” languages such as Microsoft’s Visual Basic) is fueling the shift toward component-based development.

As an example of how a shift in programming languages can affect the technologies necessary for maintenance, consider C++ and Java. These languages employ *dynamic binding*. Therefore not as much is known prior to run-time in terms of what objects can affect what other objects. Because of this, standard maintenance approaches such as static slicing are not yet viable (because of dynamic binding as well as other issues such as operator overloading, exception handling, threads, polymorphisms, inheritance, etc.).

As we continue to move toward object-oriented languages and component-based software engineering, software development will shift more towards being a manufacturing process than an inventive, creative process. It will become an assembly process that connects components that were developed by independent vendors.

“Componentware” has become a new term that refers to off-the-shelf software components that were developed by independent vendors. Componentware encompasses software subsystems that are licensed in executable format as well as software libraries that were developed in-house (*e.g.*, string processing routines, parsers, etc.).

Figure 1 shows various categories of componentware: Commercial-Off-The-Shelf (COTS), public domain, freeware, shareware, and “copyleft” software. When these types of components are incorporated into a system, maintaining the system becomes much harder. The problem is that when you maintain a system that contains unfamiliar components, or worse yet, black box components, only part of the system is truly visible to the maintainer: the part available in source code format. Because much componentware is only available as a black box, visibility during maintenance is greatly reduced. This reduces system maintainability.

For example, most applications built for Windows-NT employ Microsoft’s Foundation Classes. When you build a Windows-NT application, you have effectively teamed with hundreds of Microsoft’s developers to develop the application. But when the application needs maintenance, you *become* a one-person team.

This shift toward multi-use components forces us to rethink the technologies that we will need to maintain composite systems. If we are vendors of components, then instead of simply thinking as if we were maintaining a block of source code within a specific application, we must now think about maintaining code that is reused in each of our customers applications. Each application might have slightly different needs from our components, and thus modifications to our components may not be tolerated by all applications.

If we are component integrators, we must think in terms of technologies that will allow us to maintain the full system (that contains those components). If the components are “black boxes,” the only visibility we will have is via the documentation that describes the component’s operation and functionality. Although maintaining unfamiliar code is a com-

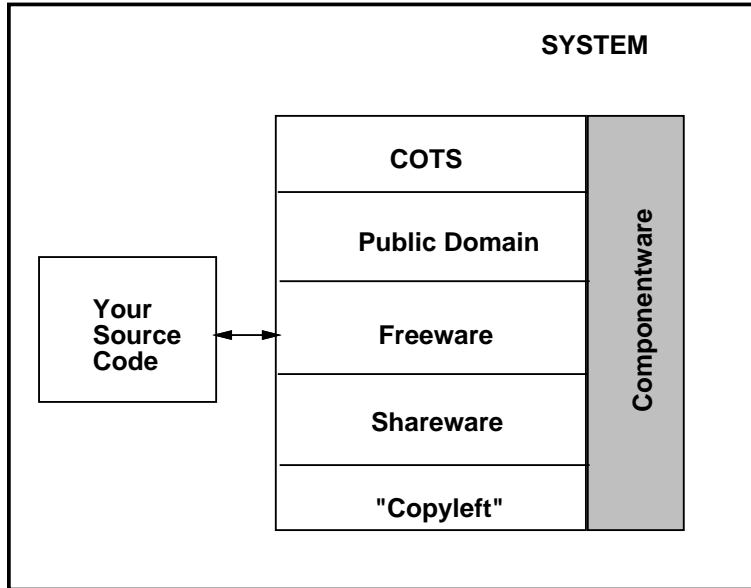


Figure 1: Composite System

mon maintenance dilemma, maintaining systems filled with black boxes adds a new level of difficulty.

Thus new software maintenance technologies are sought for component-based systems. These technologies must account for reuse across multiple applications. The remainder of this paper focuses on the new maintenance challenges associated with systems that employ:

1. Commercial-Off-The-Shelf utilities,
2. Third-party utilities (such as *freeware*, *shareware*, and *public domain* software), and
3. Reusable in-house component repositories.

### 3 COTS Software

We begin by considering systems that incorporate someone else's software that is delivered in executable format.

Maintaining these systems can be a nightmare because of:<sup>2</sup>

1. Frozen functionality,
2. Incompatible upgrades (*e.g.*, upgrades with new features or bug fixes that are not compatible with the rest of the system even though those features may be independently reliable),

---

<sup>2</sup>In this listing, we have deliberately avoided the problem of knowing whether a candidate component is the proper component for the needed task. We do not dismiss this problem as solved or trivial. Instead, we prefer to focus only on the problems that could ensue after a component is adopted into the system.

3. Trojan horses,
4. Defective, unreliable COTS software, and
5. Complex and defective middleware (*e.g.*, wrappers) that brokers information between COTS software and custom software.

We now will look at each of these situations and assess where we stand with current maintenance technologies.

### 3.1 Frozen functionality

*Frozen functionality* occurs when the vendor of an executable component either goes out of business or decides to no longer support the component. Here, applications become unmaintainable because parts of the application become frozen. If periodic updates are required for those components (*e.g.*, if the software is a parser which needs modifications each time the language changes), then the system developer has a serious problem.

This problem does not have an easy solution [8]. Options here include the user: (1) attempting to implement that functionality themselves, (2) acquiring the functionality elsewhere, and (3) acquiring the source from the current vendor and maintaining it themselves. The first option is probably implausible unless you have the required domain expertise. The second option is plausible only if there are competing alternatives.

If there are no competing alternatives and writing the functionality from scratch is unrealistic, the third alternative is the only option. To ensure you have this option if you need it, negotiate for the right to source in the licensing agreement. If you are forced to exercise this option, the problem will be maintenance personnel that do not have the proper domain expertise to maintain unfamiliar code. This could be disastrous. Hiring the people who wrote the code as consultants is a workaround here. But the cost of hiring these people could be prohibitively expensive.

### 3.2 Incompatible Upgrades

Software vendors upgrade and maintain components in accordance with the desires and needs of the greatest segment of their existing and potential customer base. If modifications or bug fixes that you need do not align with what fellow customers demand, a component that you have relied on may one day become incompatible with the remaining parts of your system.<sup>3</sup>

The problem then is similar to the frozen functionality issue: whether to abandon the vendor's component and build your own or look to competitors for options. Assuming that: (1) building your own replacement component, (2) acquiring it elsewhere, or (3) modifying your software are not feasible options, there is another technical approach that may

---

<sup>3</sup>When we talk about incompatibilities in Section 3.2, we assume that component upgrades were reliable but not acceptable to a specific application. In Section 3.4, we address the component reliability issue.

work: building *wrappers* around a component to keep the component from exhibiting those behaviors that have made the component incompatible (Discussed further in Section 3.5.)

If it turns out that component upgrades have caused incompatibilities in how a system hooks to the upgraded components, then wrapping alone may not be plausible. Here, it may also be necessary to rewrite the “glue” that connects the acquired components to the others.

If wrappers or modifications to the “glue” do not solve the upgrade problem, *downgrading* (which is sometimes referred to as *uninstalling*) may be necessary. Uninstalling can be difficult since steps taken during an upgrade are not always easily reversed.

Like uninstalling an operating system or application, uninstalling components is destined to become a very common maintenance activity. Automated “component downgrading” tools will be needed, however 100% automation may not be possible. A similar situation occurs with automated Year 2000 conversion tools, where even the best tools still only succeed in automatically fixing 90-98% of the needed conversions. The rest are handled manually.

Interestingly, the need for downgrading would go away if we could determine *a priori* whether an upgrade will be compatible. This problem is very difficult, however. Several dynamic approaches for certifying whether COTS components will be compatible in a particular system are discussed in [7].

In summary, the fear of incompatible upgrades is real. Except for the case where an upgrade is truly a fixed version, upgrades require that “tried and tested” functionality be swapped out and unproven functionality be swapped in. The moral of the story is to be ready to downgrade when compatibilities occur.

### 3.3 Trojan Horses

A software Trojan horse is intended malicious behavior that is programmed into a component in a disguised manner. For example, changing to a privileged directory and deleting all files is an example of Trojan horse functionality.

What is or is not a Trojan horse is difficult to determine *statically*. The best way to hide a Trojan horse is to make it *dynamically* context sensitive. For example, a “delete all files” command may be perfectly legal if it refers to files in a temporary directory. The same command could have terrible consequences if it changes its context to a system directory and begins removing system files. Therefore, deleting system files in this latter manner would classify as Trojan horse behavior whereas deleting temporary files would not. If the directory to which the delete is applied can only be determined at run-time, static methods will provide little help.

Avoiding swapping in an executable component with malicious behavior is virtually impossible. Nonetheless, it is a key maintenance challenge for component-based systems. Malicious behavior detection is difficult enough with source access, and nearly impossible to detect without source access [5]:

“The mass-market software industry has also seen a problem with logic bombs and Trojan horses. For example, in 1994 Adobe distributed a version of a new

Photoshop 3.0 for the Macintosh with a ‘time bomb’ designed to make the program stop working at some point in the future; the time bomb had inadvertently been left in the program from the beta-testing cycle. Because commercial software is not distributed in source code form, you cannot inspect a program and tell if this kind of intentional bug is present or not.”

Unfortunately, detecting malicious behavior in an executable component requires monitoring all requests from the component to the operating system as well as checking the context of the request. For components, the place to do this is in a wrapper around the component (See Section 3.5). Note, however, that this can be an enormous number of calls and context checks, and the likelihood of avoiding false-positives and false-negatives is low. That is, the wrapper will filter away some that it should not, and will not filter away others that it should have.

In short, Trojan horses in executable components are virtually undetectable. This is not an issue that the software maintainers can easily solve, but it is an issue that they must be aware of when they swap in new components during maintenance.

### 3.4 Unreliable COTS Components

Section 3.2 discussed incompatible upgrades. Here we focus on the problem of unreliable COTS components. The two problems are related but distinct.

Today, no uniform standards exist by which software components are tested and certified for reliability. There are, however, process measurement schemes (*e.g.*, CMM, ISO, etc.). Good processes do not guarantee good software [6], and for COTS software, even if good processes did, a vendor could lie about their process maturity.

Software reliability models have been proposed for years [1]. But the assumptions that these models often make about environments, defect rates, defect severities, fault sizes, etc., are generic. These assumptions may not reflect the idiosyncrasies of different environments.

Thus a component’s dependability is unknown to its potential adopter. And even if a dependability score were provided by a component’s vendor, it is possible that the score was based on peculiarities that do not reflect the adopter’s environment.

If maintenance is to someday become the process of swapping components in and out, then there must be a universally accepted way to assess the dependability of a component. And this approach must provide enough information to account for environmental variability. We have standard ways to assess quality for transistors and price them accordingly. Why don’t we for software? It would certainly make component-based maintenance less of a gamble. And it would allow us to know better how a system is going to behave before a swap is made.

### 3.5 Middleware: Wrappers

Middleware is software “glue” that hooks components together. Middleware is designed to semantically or syntactically modify how components interact. Whenever there is doubt about how a component will operate inside of a system, writing middleware to enforce certain design constraints is prudent.

*Wrappers* are a form of middleware that limit the functionality of components. Wrapping technology has roots in the safety-critical community which has long used techniques to partition safety-critical functions from non-safety-critical functions. The main ways by which wrappers work are to either: (1) restrict the inputs, or (2) restrict the outputs of a component. Both have the same effect of altering a component’s semantics.

The most fundamental problem with wrappers is knowing what behaviors to protect against. Behaviors may be unknown to the maintainer and thus not adequately protected against. Querying the vendor could shake lose some information, but the best approach will be to test a component as heavily as possible in the system environment. By combining vendor information requests and in-house testing, more thorough wrappers will be possible.

In summary, wrappers are a reasonable means for addressing the aforementioned incompatibility, Trojan horse, and dependability problems. But wrappers are not foolproof. They can be complex, incomplete, and unreliable. And if you do not suspect that a particular behavior will need to be protected against, you are unlikely to build a wrapper that does so.

## 4 Third-Party, Non-Commercial Software

So far, we have discussed maintenance issues for systems that employ executable commercial software (COTS). In this section, we will briefly discuss issues related to maintaining systems that employ third-party components. The types of components that we are interested in are: shareware, freeware, public-domain, and “copyleft.”

*Shareware* is usually freely down-loadable evaluation software. The user is morally urged to send a nominal distribution fee (*e.g.*, \$20) back to the originator if the user wishes to continue using it. An example here is Nico Mak Computing’s WinZip. If you do not register and pay the fee, shareware usually times out. Usually, source-code is not distributed with shareware, and thus all of the same maintenance problems already discussed with respect to COTS software apply here as well. Similar to the COTS Trojan horse problems, Garfinkle and Spafford [5] state the following concerning shareware:

“Like shrink-wrapped programs, shareware is also a mixed bag. Some shareware sites have system administrators who are very conscientious, and who go to great pains to scan their software libraries with viral scanners before making them available for down-load. Other sites have no controls, and allow users to place files directly in the down-load libraries. In the spring of 1995, a program called PKZIP30.EXE made its way around a variety of FTP sites on the Internet and

through America Online. This program appeared to be the 3.0 beta release of PKZIP, a popular DOS compression utility. But when the program was run, it erased the user's hard disk."

*Freeware* is similar to shareware except that it is totally free. Anyone can use it. An example here is Duncce (a Win95 dialer add-on). Freeware is usually distributed in executable format only. Thus the aforementioned COTS maintenance problems reappear here.

*Public-domain* software is freely distributed in source-code format with no restrictions on its usage. It can be modified at will. For example, Sun Microsystems used BSD Unix (public-domain) to build their proprietary operating systems. Maintaining public-domain software or using it to evolve a new family of software products is not trivial, however. This incurs the same problems already discussed with respect to maintaining source code from a vendor that has opted to no longer support the code. (Discussed in Section 3.1.)

There is a slightly different form of public-domain software that is referred to as "GNU/FSF *Copyleft*." Copyleft software has licensing restrictions on how it can be used. For example, GNU's source must always retain the GNU copyleft notice and it is illegal to embed it into proprietary applications. Copyleft software suffers from all of the aforementioned problems of maintaining unfamiliar source.

In summary, many of the maintenance options for free or almost-free third-party software will depend on licensing restrictions. If the software is delivered in executable format, the COTS problems apply. If source code is provided, then the problem of having enough domain expertise occurs.

## 5 In-House Component Repositories

We have addressed maintenance issues surrounding systems built from acquired componentware. In this Section, we will focus on maintaining in-house components stored in proprietary reuse repositories (libraries). This type of software is represented by the box that says "Your Source Code" in Figure 1.

There are important differences between: (1) maintaining the source in components that are only used in one system, and (2) maintaining the source in components in a reusable library. The main difference is that changes to components in the one-of-a-kind system only impact that system.

The first challenge in maintaining components stored in in-house repositories is ensuring that component modifications are compatible to all applications that use the component. Recognize that reusing a component in multiple applications often constrains the specification of the component to a common set which may not satisfy the needs of all applications.

Software repositories employ different access rules to address this problem. By adding logistical rules on how components in a repository can be modified, software repositories can better organize software development and maintenance teams.

However, developing software according to repository access rules can stifle developer creativity since developers are required to: (1) check source out prior to modifying it, and (2) test the changes prior to integrating the software back into the library. For developers these steps are burdensome. But from a maintenance perspective there must be *revision control* or else the component maintenance process will break more applications than it will aid.

The architecture of a component repository can affect the maintenance process for all systems that incorporate repository components. *Functionally* structured repositories do not lend themselves for ease of maintenance. Applications can find themselves coupled to everything in the repository and not just a particular component's information. Functionally structured repositories are oriented around the types of information that they contain. For example, assume that we have three components X, Y, and Z, each of which stores the following types of information: analysis, design, source code, and test. A functionally structured repository would look like:

```
source:  X
         Y
         Z

analysis: X
         Y
         Z

design:   X
         Y
         Z

test:    X
         Y
         Z
```

Here, extracting *only* X, Y, or Z during maintenance is difficult. To extract X, Y, or Z requires extracting X, Y, and Z. And extracting everything from a repository requires massive amounts of storage rendering this approach as impractical.

Now consider an *information class* repository structure:

```
X: analysis
   design
   source
   test

Y: analysis
```

design  
source  
test

Z: analysis  
design  
source  
test

This architecture allows extracting only X's information without extracting Y or Z's information. Component X can be maintained more easily in this architecture.

One problem with requiring component maintainers to check out components from a repository using revision control software is that the components may become exclusively locked for long periods of time before the changes are committed back into the repository and the components are unlocked. File locking compromises the maintenance of other applications that use common source components. But if file locking is not used, other problems ensue. For example, a developer may make such substantial changes to a component that merging those changes with someone else's changes becomes a nightmare.

A different approach to maintaining components in reuse repositories is called "promotion." Promotion has several levels which are not oriented to particular applications but instead to all applications. The levels are: (1) development/maintenance, (2) test, and (3) release.

With the promotion repository architecture, maintainers checkout component source from the development/maintenance level, make changes, and commit the changes to their own personal repositories. When confident that the changes are correct, they promote their code to the development/maintenance level where a manager controls the integration of the changes prior to promoting it to the test level. At the same time that this is happening, developers are still able to check out component source from the development/maintenance level and make additional modifications. After testing approves the changes, the development manager will promote the software to the release level. Once at the release level, it is made available for general use.

Thus, in summary, a repository's architecture is vital to the successful maintenance of systems built from in-house components. Because in-house components are frequently deployed in different applications, revision control is imperative and a reasonable repository architecture can make revision control less invasive to the creativity of software developers and maintainers. Other necessary activities for maintaining in-house components are testing, documentation, and traceability back to a component's specification.

## 6 Summary

Component-based software development is founded on the principle known as divide-and-conquer. Because in practice, much of this paradigm relies on software units that were developed elsewhere, traditional software maintenance solutions need to be rethought and upgraded.

Without a doubt, the best advice for component maintainers is to avoid growing components into mini-systems. Be careful to keep detailed requirements documentation on each component, and avoid the temptation to keep endlessly adding “bells and whistles.” Use an information class repository structure and promotion, and don’t hesitate to have two similar components in a repository if competing applications cannot tolerate changes made to the component for each other.

Traditionally software maintenance has been an expensive process involving impact analysis, regression testing, and much manual effort. In the future, however, software maintenance may be as simple as swapping components in and out. But as we learned from the Ariane 5 disaster, even reusing components from environment to environment is not safe [3]. Integration testing is an important activity that must be done as a part of maintaining reusable components. And this testing must represent the peculiarities of the application [7].

Component-based software engineering may someday afford us the ability to rapidly produce information systems. The downside is that these systems are much harder to certify and validate than their predecessors. If the certification and validation issue is not adequately addressed, these systems may be unmaintainable since legacy systems require frequent regression testing.

We are not yet at the point of easily swapping components in and out as a means of system maintenance. Further, we are not yet to the point of having software catalogues with competing components like we have with other retail products. If we do get to the point of having component catalogues, a new set of maintenance issues will occur: building correct wrappers, detecting malicious behavior, equating specifications between what is needed and what is offered, avoiding frozen functionality, detecting unreliable components, performing COTS testing, having access to independent component dependability certification [9], and detecting *a priori* whether component upgrades are compatible. Cars, airplanes, and buildings are maintained via replacement parts and have overcome these challenges. Why not software?

## Acknowledgments

I’d like to thank the following colleagues for commenting on earlier drafts of this manuscript: Lora Kassasb, Chuck Hutchison, Chuck Howell, and Rich Mills. Voas has been partially supported by DARPA Contracts F30602-95-C-0282 and F30602-97-C-0322, National Institute of Standards and Technology Advanced Technology Program Cooperative Agreement Number 70NANB5H1160, US Army Contract DAAL01-98-C-0014, NASA-Ames Contract

NAS2-98052 and Rome Laboratories under US Air Force Contract F30602-97-C-0117. THE OPINIONS AND VIEWPOINTS PRESENTED ARE THE AUTHOR'S PERSONAL ONES AND THEY DO NOT NECESSARILY REFLECT THOSE OF THESE AGENCIES.

## References

- [1] C. V. RAMAMOORTHY AND F. B. BASTANI. Software Reliability - Status and Perspectives. *IEEE Transactions on Software Engineering*, SE-8:543–371, July 1982.
- [2] COMPUTER NEWS DAILY. "Year 2000 Prophet Preaches \$600 Billion Digital Fix", October 1, 1997.
- [3] Prof. J. L. LIONS. Ariane 5 flight 501 failure: Report of the inquiry board. Paris, July 19, 1996, available at [http://www.cnes.fr/actualites/news/rapport\\_501.html](http://www.cnes.fr/actualites/news/rapport_501.html).
- [4] Personal discussions with Sam Keene at Hughes.
- [5] S. GARFINKEL AND G. SPAFFORD, editor. *Practical Unix and Internet Security, Second Edition*. O'Reilly and Associates, Inc., 1996.
- [6] J. VOAS. Can Clean Pipes Produce Dirty Water? *IEEE Software*, 14(4):93–95, July 1997.
- [7] J. VOAS. An Approach to Certifying Off-The-Shelf Software Components. *IEEE Computer*, To appear in June 1998.
- [8] J. VOAS. COTS Software: The Economical Choice? *IEEE Software*, 15(2):16–19, March 1998.
- [9] J. VOAS. Software Certification Laboratories: To Be or Not to Be Liable? *Crosstalk*, 11(4):21–23, April 1998.
- [10] M. WEISER. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.