

## Chapter 1

# USER PARTICIPATION-BASED SOFTWARE CERTIFICATION

Jeffrey Voas

**Abstract** Except for a couple of rigorous software certification schemes that are required for certain safety-critical software applications (e.g., RTCA DO178-B), there are no generally trusted software certification schemes. While there is no theoretical reason why these safety-critical standards could not be applied to all applications, there is a practical reason: cost. Estimates vary, but it is generally accepted that a line of safety-critical software costs between \$500-\$5,000 to design, write, test, and certify. Such levels of rigor and quality are excessive for most non-safety-critical applications.

Nonetheless, a ubiquitously accepted process which provides guarantees about software quality is still needed. These guarantees must be product-based and trustworthy. This paper presents a certification process we believe satisfies this need. Our process collects appropriate data that can then be used to stamp limited warranties onto commercial software.

**Keywords:** certification, software warranties, residual testing, software users, software publishers

## 1. INTRODUCTION

Today, the number of approaches and standards for certifying software quality is increasing. These approaches either validate the integrity of the software product, development processes, or personnel [7].

The most popular approaches are process-based (e.g., ISO9000 and SEI-CMM). Here, software publishers usually take oaths concerning which development standards and processes were used. Also, auditors may be employed to “spot check” a publisher’s project documentation and oaths [7]. Even if a certification auditor can verify that the software publisher was truthful, that alone does not guarantee high quality software. An analogy here is that dirty water can run from clean pipes [5].

We propose a certification methodology that does not employ auditors and publisher oaths. We believe that completely *independent* product certification is the only approach that consumers should trust. And demands for access to *independent* agencies that can play this role are being heard from both publishers and end-users.

Publishers prefer independent agencies since they then are not responsible for warranting their own software. By hiring a third party to grant software *certificates* (or *warranties*), publishers shift this responsibility onto someone else (much like when a doctor orders a second opinion or another test). End-users also benefit from independent agencies that offer unbiased assessments. Therefore the business case for creating independent agencies to certify software quality is strong.

We will refer to agencies that perform third-party, independent software certification as Software Certification Laboratories (SCLs). The beauty of having independent SCLs is that they provide a fair “playing field” for each publisher (assuming that each product under review receives equal treatment). A key reason, however, that SCLs have not become widespread can be attributed to the liability of being a certifier. When certified software fails in the field, the certifier bears some level of liability.

Why? Because SCLs represent themselves as experts. Courts in the United States are notorious for holding persons and organizations that represent themselves as professionals to unusually high standards. For example, suppose a surgeon, who never erred after 999 operations, makes a serious, *negligent* mistake during the 1,000th operation. The surgeon has a failure rate of 0.001. Even though the surgeon is highly skilled, very reliable, and has helped 999 patients, the surgeon could still be sued into bankruptcy because of one case of negligence.

SCLs bear a similar *liability* with similar consequences for mis-certification [6]. In order to reduce an SCL’s liability, accurate methods for making certification decisions must be employed. Unfortunately, even the best static analysis techniques and testing techniques often fail to consider the actual stresses that software will experience when in the hands of users. Thus SCLs suffer from the problem of accurately determining how well-behaved a software system will be in the future. And that is the key piece of information we need from certifiers.

The certification process we will propose greatly reduces this liability quagmire while also eliminating the need to dispatch human auditors. Our process harnesses the testing resources of end-users. It is similar to the way in approach that made Linux the most popular and reliable of all Unixes [2, 4].

## 2. OUR SOFTWARE WARRANTY MODEL

Our interest is in certifying software applications and components with dual-use potential. We are interested in certifying applications such as Microsoft Word. And in fact, we are more interested in smaller components that could be embedded in a variety of applications. Our goal is to provide a certification process that fosters greater software reuse throughout different classes of applications.

Component-based software engineering (CBSE) is simply building software systems from software parts. By building from software parts, reuse is increased. This has the potential to substantially decrease the cost of development and decrease time-to-market.

But the virtues of CBSE have been touted for years. And there is *little* evidence that we are close to CBSE becoming the de facto standard for how software systems are built. Why is this when glue technologies (e.g., CORBA, ActiveX, COM, DCOM) for component interoperability and reusable component libraries already exist?

In our opinion, CBSE has not become ubiquitous because of: (1) widespread distrust of software components, and (2) a lack of knowledge about how to design-for-reuse. Any two components can be glued together rapidly, but that is not sufficient for popularizing CBSE. Integrators must have confidence that the right components were glued, they were glued together correctly, and the components are dependable. (Also, integrators must be able to find the components that they need, but we will assume that component visibility will eventually become a solved problem with the World-Wide Web.)

And as already mentioned, trust in a component's dependability must come from someone other than the software publisher. We have already listed two reasons for this: the liability associated with making claims concerning quality, and end-user distrust of publishers. Two additional reasons for not allowing the software publisher to be the person to assess the goodness of their products are: (1) software publishers are unlikely to have adequate resources to test to the levels that would justify software warranties, and (2) publishers are likely to make incorrect assumptions about how users will use the software.

Interestingly enough, SCLs that might hope to employ in-house testing as a means to certify software will suffer the same problems just mentioned: inadequate levels of testing and incorrect assumptions. In fact, the degree of testing that can be performed in a cost-effective manner at a SCL may be less than that already done in-house by the software publisher. Thus the idea then that SCLs can do in-house testing as a means for granting warranties is dubious.

In our opinion, software warranties must be the end-result of massive amounts of operational testing. Such testing will have demonstrated product stability in fixed environments and fixed marketplace sectors (embedded,

desktop, Web-based, etc.). Without this, these limited warranties will either be too restrictive or unbelievable.

If SCLs and publishers cannot perform adequate product testing, who can? Our answer is the disorganized body of software testers known as the users. They can be unified to overcome this problem and do so in a way that is advantageous to them. The issue then is how to best tap into them in a manner that makes software warranties into a reality.

We propose user-based product certification as a means for granting software warranties. Our approach collects valuable field data in a non-intrusive manner from the user's environment with their permission. Currently such data is rarely collected.

Our approach will exploit testing technologies similar to the *residual testing technologies* discussed by Pavlopoulou [3]. Residual testing is post-release testing that tests fielded software as it operates. Residual testing employs instrumentation that is embedded in operational software to collect information about how the software behaves. This information can reveal what software code is executed, whether assertions fail, and whether the software itself fails (e.g., if a limited regression testing capability is bundled with this instrumented version). We will also collect information on how the product is used. We can measure the frequency with which certain features are called, simply collect files of inputs (that later can be used to build operational profiles), etc. These sets of information will serve as the basis for issuing *limited* software warranties.

Our certification process is shown in Figure 1.1. Here, a software publisher subjects their finished product (i.e., the post- $\beta$  version) to a residual testing tool which creates a fixed number of identical, instrumented copies (non-identical copies could also be built but that will make the tasks of the SCL more complicated). The copies are then supplied to the SCL. The SCL provides the instrumented versions to pre-qualified users from different marketplace sectors. Pre-qualified users will be those who will use the product in a manner consistent with how the SCL wants the product used. (Microsoft's beta-user program is an excellent example of how to screen possible users to those who will actually provide the greatest amount of information.) Periodically, the SCL gathers the information from the user sites.

All user information is merged and statistics are computed by the SCL. The SCL provides: (1) statistical data back to publishers concerning how their product was used, and (2) data telling publishers how their products behaved in the field (i.e., product quality). Statistics must be generated such that a backwards trace to any specific user is impossible. This is of paramount interest. (We will later look at why previous attempts by publishers to force users to reveal similar information were unsuccessful and how our model avoids those pitfalls.)

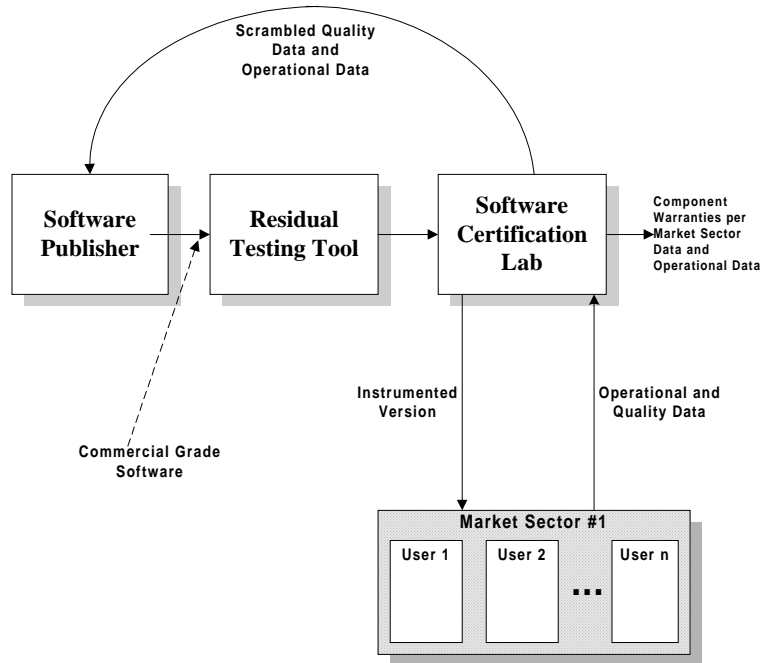


Figure 1.1 Basic Certification Process

Recall that the goal of this process is for an SCL to confidently determine whether a product should be warrantied and what it should be warrantied for. That is, which platforms, which market sectors, and which environments? This is why we refer to our warranty as “limited.” As an SCL gathers additional data over time, the SCL may be in a position to broaden a warranty, i.e., make it into a fuller, less restrictive warranty.

The motivation for our approach to software warranties stems from a client request some years back. They wished to employ the AIX operating system in a highly safety-critical environment. The client needed to know how reliable different AIX operating system and library calls were. The client could not find the information (nor did IBM know) even though thousands of AIX users worldwide had generated enough anecdotal evidence that when taken together, almost certainly would have answered our client’s question. But because that information was lost, the client had to spend substantial resources re-testing each operating system and library call. And worse, it is possible that other organizations are redoing what our client did. This is clearly wasteful.

Note that we are not suggesting that you seek volunteers to fly uncertified software-controlled aircraft or use uncertified software-controlled medical devices in order to see if the software in them should be certified. Instead, what

should occur is for the software to be first certified for non-critical market sectors. Then, if a supplier of a safety-critical product were to discover that this certified software utility was certified for an environment that mirrors the safety-critical product's environment, then the supplier has the option to employ it. (Thus all we have said here is that the operational profiles are equivalent between the non-critical and safety-critical environments.) If the software behaves well in this and other safety-critical applications, and if enough of this evidence can be collected demonstrating confidence in the software utility, then evidence will exist that warrants certifying the utility for the safety-critical market sector. Clearly much caution must be taken here, but recall that we are dealing with post- $\beta$  software to begin with. We are never dealing with untested software, even when a piece of software is first embedded into a safety-critical environment.

### **3. KEY CHARACTERISTICS OF OUR MODEL**

We have described our model. For our model to be acceptable to users, SCLs, and publishers, we deliberately built in the following characteristics:

1. Certification is performed on a stable, non- $\beta$  version of the product.
2. Certification is not done by the software publisher. The publisher delivers a fixed number of versions of a product to the SCL. These versions have residual testing capabilities built-in. The SCL licenses the versions to pre-qualified users from various application domains (e.g., safety-critical, desktop, Web, etc.).
3. The testing methods and type of usage information collected must be determined by the SCL. This information should also be made available to users.
4. Certification decisions are based on significant operational experience created by the user base. This approach employs orders of magnitude more testing than the publisher or SCL could perform. Also, certification decisions are based on real operational scenarios as opposed to hypothesized scenarios. Hypothesized scenarios are usually the best that a SCL and publisher can base pre-release testing on. That will rarely be good enough, for example, to convince an actuary in an insurance company to insure a software product. Because our approach employs field data, the data an SCL will collect could be used by insurers that hope to insure software-based systems. Also, insurers may feel confident enough to indemnify an SCL against mis-certification since each warranty is based on historical data.

5. User data is collected by background processes that minimally interfere with users. Users expend no resources to trap and collect the information (other than the computer resources necessary to gather and store the information). Although users are not required to perform manual tasks to collect the information, their versions will have performance degradation due to the background processing cycles required to perform the post-deployment testing. Users that participate in this accelerated testing program will be compensated with reduced rate or free software.
6. All testing data collected from users is delivered exclusively to the SCL. The raw information does not go to the publisher. SCLs agree to legal confidentiality between themselves and users. Once the raw information is scrubbed and user identities are not traceable from the composite information, the composite information is passed back to publishers. This allows them the opportunity to improve quality on future upgrades and focus in-house testing toward their user base.
7. A software warranty is limited to different market sectors (safety-critical, desktop, e-commerce) and different environments (Unix, Windows-NT, Macintosh). When enough data is collected from the field affirming that a component is working properly in a particular market sector, the SCL will provide software warranties specific to that market sector. For example, a SCL warranty might read as: software product  $X$  is warrantied to perform with a reliability of 99.9 in the Windows-NT desktop environment.
8. Users *consent* to participate. Users that do not wish to participate in this process do not have to. They are free to license non- $\beta$  versions that do not undergo residual testing.
9. Publishers compensate SCLs for their services. The SCL will also own the collected data. This data has economic value. The laboratory will reserve the right to sell the data (possibly in a format like that found in *Consumer Reports* magazine). Recall our AIX story where a client had to re-test the software to get the data it needed on AIX's dependability, and other organizations will have to do the same. Why not create the information once and then sell it? Also, it is plausible that a non-profit company or government agency serve as an SCL.
10. No auditors are needed. Most existing certification models require that trained auditors visit publisher sites and sift through requirements documents, test plans, etc. Our scheme avoids the pitfalls associated with human auditor error.
11. Unlike ISO-9000 and CMM, our product-based certification allows innovative processes to emerge. If a developer organization can produce

certifiably good products that conform to an in-house standard, then they have developed new processes that other development organizations should consider. After all, ISO and CMM may lock us into old process ideas prematurely.

12. And finally, this process forces publishers to define what is correct behavior for their software. This alone could result in higher quality software products.

Clearly there are benefits to our approach. But those benefits cannot be realized until we know precisely what data is sufficient to justify issuing software warranties. We envision it including reliability assessments, data from assertions, monitoring exception calls, data on operational usage, and code coverage analysis. But ultimately it depends on how strong of a warranty is sought.

Other analyses are possible, but these seem to be the more important indicators of whether a component has been exercised thoroughly enough to warranty it for specific environments and market sectors. This is a topic for future research. Our immediate plans are to work with technology insurers to select the residual testing technologies that provide data sufficient for offering software insurance premiums and software warranties.

#### 4. RELATED APPROACHES

While our model for combining SCLs with residual testing technologies is unique, it admittedly leverages existing and previous ideas from other organizations. Here, we will compare our model with these other ideas.

Our first example is from a product called PureVision that was released in 1995. PureVision was offered by Pure Software. The product performed crude residual testing functions. It worked in a manner similar to how we have defined residual testing: a publisher produced copies of a product that were able to monitor themselves at user sites [1]. The copies sent back a report *to the publisher* concerning: (1) which user and user site were using the product, (2) the version number, (3) system configuration, (4) when the version started executing and stopped, (5) which program features were used, and (6) the amount of memory used at exit. If the product failed, exit codes and a stack dump were added to the report. Pure knew that users would be wary of publishers looking over their shoulders and thus included an option whereby a user could inspect a report before it was sent back. An option to not send a report back was also available. According to former Pure employees, speculation for why PureVision did not survive is that users were unwilling to provide such detailed, non-technical information (e.g., which user was using it, at what times, on which host, etc.). In contrast, the information we will collect is more technical; our information spies on the software and its correct behavior as opposed to spying on the user.

Our second example is similar to PureVision. Netscape 4.5 contains an option called The Netscape Quality Feedback Agent. The agent sends feedback to Netscape's developers concerning how the product is performing. The agent is enabled by default and is activated when Communicator encounters some type of run-time problem. When the agent is activated, it collects relevant technical data and displays a form in which a user can type comments. Netscape uses this data to debug known problems and identify new ones. Unfortunately, however, most users do not activate this feature for similar reasons to those for why PureVision did not survive.

Our next example is taken from the Software Testing Assurance Corporation (STAC) which was a for-profit venture started in 1997. STAC attempted to be the sole certifier for Year 2000 insurance on behalf of insurers. STAC's certification procedure was based on a public-domain testing standard (for Year 2000 remediated code) and STAC-approved auditors who were to make site visits to companies seeking Year 2000 software insurance. In the end, however, these insurance policies were never offered because insurers felt that the assessment approach for whether the software had been remediated correctly was too risky.

Microsoft has long employed beta-testing as a way to collect information on how their products perform in the hands of users. And for this information Microsoft compensates users. Pre-qualified users are given advanced copies of a product at reduced rates in exchange for feedback concerning product stability, usability, and reliability. Microsoft uses this information to decide when a product is ready for general release. This demonstrates user willingness to participate in exchange for discounted software.

And finally, consider Linux, a Unix operating system project that began in 1991 and today has nearly 8 million users. Linux is the product of hundreds of users, all of whom donated their time to write the system. Their efforts paid off: Linux is considered to be the most reliable of all Unix operating systems [4]. In fact, the success of Linux is often used as the argument for why products with open source are the only products that are trustworthy.

So what can we surmise from these anecdotes? From Microsoft and Linux, we see that users are willing to participate in efforts that result in improved software. From PureVision and Netscape, we see that publishers have a serious desire to attain field information. And from STAC, we see interest from industry in forming for-profit software certification corporations.

What still remains to be seen, however, is whether users will trust an SCL to act as an intermediary between themselves and a software publisher. As long as the confidentiality agreements between: (1) the SCL and publisher, and (2) SCL and user are binding, we believe users will.

## 5. SUMMARY

Demands for SCL services translate into business opportunities for those that can overcome the liability risks. In fact, several SCLs are already in existence today. For example, KeyLabs certifies language purity for Java but makes no promise about software quality. Because language purity is trivial to test for, the liability for KeyLabs is low.

Our approach dissolves the distrust associated with schemes that use auditors or measure process maturity. We capture user information that is traditionally discarded: (1) is the software behaving correctly?, and (2) how is the field using the product? Our approach does not invade user privacy; users agree to license instrumented versions from intermediaries who sit between them and the publishers.

Our approach has the potential to decrease the cost of software development by fostering CBSE. If software applications can be warrantied under certain environments, then users with equivalent environments no longer need to guess at how a commercial product will behave. Nor do they need to invest heavily in re-testing the application.

And finally, the time it takes to get a limited software warranty can be reduced by offering more instrumented versions or only releasing instrumented versions to persons that are committed to seriously stressing a product. This will reduce the time until all users can enjoy un-instrumented, warrantied software.

## Acknowledgments

This work has been funded by Reliable Software Technologies in Sterling, VA, USA (<http://www.rstcorp.com>).

## References

- [1] Bingley, L. PureVision: Shedding Light on Black Art Betas, APT Data Services, Inc, New York, June 1995, page 17.
- [2] Miller, B. P. et. al. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Computer Sciences Dept, November 1995.
- [3] Pavlopoulou, C. Residual Coverage Monitoring of Java Programs, Master's Thesis, Purdue University, August, 1997.
- [4] Miller, B.P., Fredrikson, L., and So, B. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.
- [5] Voas, J. Can Clean Pipes Produce Dirty Water?, *IEEE Software*, 14(4):93–95, July 1997.

- [6] Voas, J. Software Certification Laboratories: To Be or Not to Be Liable? *Crosstalk*, 11(4):21–23, April 1998.
- [7] Voas, J., The Software Quality Certification Triangle, *Crosstalk*, 11(11):12–14, November 1998.