

Deriving Accurate Operational Profiles for Mass-Marketed Software

Jeffrey Voas (jmvoas@rstcorp.com)

Reliable Software Technologies, Sterling VA USA

Industries such as telecomm and avionics have long enjoyed a perk that most other industries do not enjoy: access to accurate operational profiles. Most other industries are left hypothesizing about how they believe the majority of their users will interact with a particular software package. If these guesses are wrong, reliability predictions based on the guesses will likely be wrong as well. Further, estimates as to when testing should halt and product release occur are also likely to be inaccurate.

This paper presents a model that builds accurate operational profiles for mass-marketed software. This paper also proposes a new definition of operation profile, one which better reflects all external factors that can cause mass-marketed software to fail. The key benefits to publishers from having access to accurate operational profiles are: (1) detecting “bloatware”, misused, and unused features, (2) discovering which machine configurations most users execute the publisher’s software on, (3) learning how users change usage habits when new releases occur, (4) knowing more accurately how to test their products between releases, and (5) determining how to better educate their users through improved user manuals. Put simply, our model provides publishers with an unprecedented amount of information detailing field usage. This will allow them to make better long-term business decisions concerning consumer feature preferences. Publishers armed with such information will enjoy a competitive advantage.

1 Introduction

The difficulty surrounding the generation of accurate operational profiles has long been a source of frustration in the software reliability and software testing communities. Accusations have arisen that except for the avionics and telecomm industries, accurate operational profiles are rarely unattainable. If true, this means that effort spent seeking them for other types of software will likely be in vain, and even if they are created, they are likely to be wrong [6].

Why then are the telecomm and avionics industries able to succeed at this while others fail? The answer is a wealth of historical data. Consider first the telecomm industry. They have been able to collect profiles from thousands of “user years” over decades of calendar

time. In fact, it is rumored that AT&T has on file every phone call that has been made in the last 30 years. If true, AT&T has a perfect operational profile. And the same is true for aircraft. After nearly 100 years of building aircraft, there are few operational anomalies that are not known to aircraft manufacturers. Therefore they know what needs to be done to build and fly a plane. But for software developers of new, shrink-wrapped products, deriving accurate operational profiles is not so easy.

Another problem facing publishers of mass-marketed software stems from the accepted definition of operational profile. According to classic software reliability references [4, 3], an operational profile is defined as follows:

An operational profile is the set of input events that the software will receive during execution along with the probability that the events will occur.

The problem, however, is that this definition is too narrow in scope for the bulk of the software in use today (i.e., mass-marketed, shrink-wrapped). For this type of software, the above definition needs to also take into account entities other than just the primary (usually the human) user of the system. The definition needs to also consider the operating system and other applications competing for resources. After all, it is these entities which can cause an application to fail even under the most gentle use from a human user. Further, the amount of memory accessible to the software application and other platform-related artifacts must also be included in the operational profile definition. Therefore, we recommend a definition more like the following:

An operational profile is: (1) the set of input events that the software will receive during execution along with the probability that the events will occur, and (2) the set of all input events generated by external hardware and software systems that the software is expected to interact with during execution.

In fairness to my colleagues who have provided the community with the current, more widely accepted definition of operational profile, I should mention that the problems with the definition are mainly isolated to mass-marketed software. The current definition appears to work fairly well for embedded software. The reason appears to be that embedded software is already customized for specific hardware environments (fixed memory, fixed disk space, fixed processor speeds, etc.). Thus these external hardware and software subsystems have already been factored into the embedded software's design, and thus it is acceptable to simply consider the operational profile as the set of input events and their probabilities.

But for software that is expected to run on a seemingly infinite set of hardware configurations, these factors have not been adequately accounted for in the first definition. Therefore, we will provide a model for building accurate operational profiles for shrink-wrapped, mass-marketed software. And our model will employ the latter definition of operational profile.

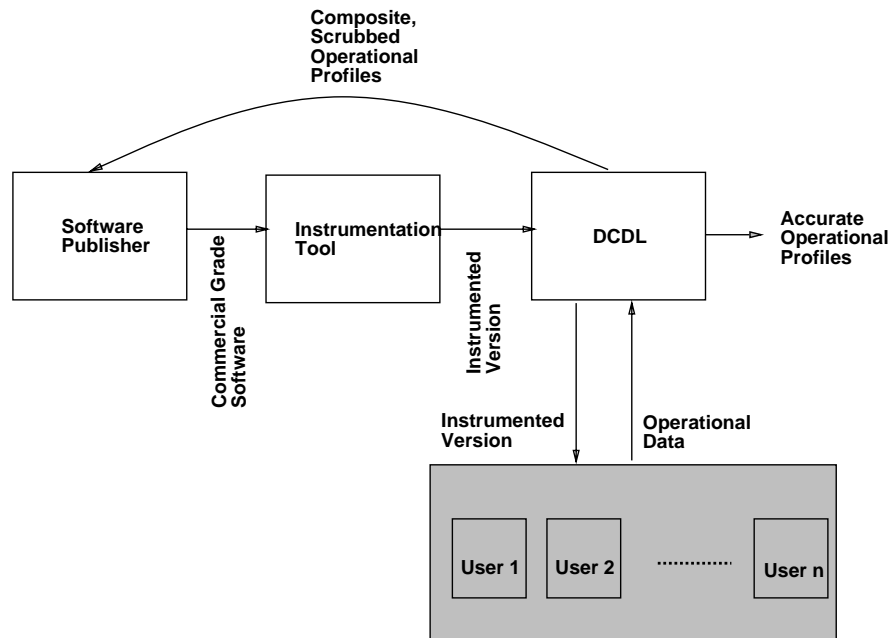


Figure 1: Operational Profile Generation Process

2 The Model

The question, however, is how can a publisher of mass-marketed software get accurate operational profiles? Software versions change quickly. One year of calendar time is 7 years of Internet time. And there is a seemingly infinite set of hardware configurations to consider. Unlike AT&T (that can store calling records for decades), software publishers of mass-marketed products have little or no historical data to fall back on.

Our answer to this dilemma is the disorganized group known as “the users.” As it turns out, they are uniquely qualified to help here and can be unified to solve the problem. The real issue then is how to organize them.

My proposed solution revolves around an information mining scheme that collects field usage data from users’ environments. Currently such field data is not collected and thus lost. Our solution will be implemented in a non-intrusive manner via automated software processes. All information is collected with user consent.

To implement this idea, we employ source code instrumentation that reports back what software features are used, in what combination, and other specifics about the machine environment. To report which features or combinations thereof are used simply requires a code coverage tool that is capable of instrumenting for feature coverage as opposed to statement or branch coverage. To collect detailed information about a user’s environment, say for a Microsoft Windows machine, there are existing utilities that already provide most of what we need: `regedit.exe` and `regedt32.exe`.

The process that we contend can lead to accurate operational profiles for shrink-wrapped software is shown in Figure 1. Here, a software publisher subjects the finished product (i.e.,

the post- β version) to an instrumenter who creates an instrumented copy of the commercial software. This version will collect all feature usage and external environment data. The version is then supplied to a “middleman” organization which we will term as the Data Collection and Dissemination Lab (DCDL). The DCDL can recopy the version numerous times based on the number of users that are willing to use that version of the product.

The DCDL provides the instrumented versions to pre-qualified users. 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 DCDL collects usage information back from user sites.

The data collected from these versions will be “scrubbed” and combined with the data collected from other users before it is returned to a publisher. The statistics used for the operational profiles will be generated in such a manner that a backwards trace to any specific user is impossible. This is of paramount concern.

3 Key Characteristics of Our Model

For our model to be adopted by users, DCDLs, and publishers, certain safeguards and characteristics have been deliberately built-in. We will now list those that we feel are most noteworthy.

1. Profile collection is performed on a stable, non- β version of the product.
2. Profile collection is not done by the software publisher. The publisher delivers a fixed number of instrumented versions of the product to the DCDL. The DCDL licenses the versions to pre-qualified, willing users.
3. The type of usage information collected will be decided by the DCDL. The onus is placed on the DCDL to ensure that the instrumentation embedded by a publisher was legitimate.
4. Information returning to the DCDL will be encrypted. The DCDL will need to validate that returning information indeed came from a legitimate site
5. Details concerning the type of information being collected must be made available to participating users in order to diminish concerns regarding privacy.
6. User data is collected by background processes that minimally interfere with users. Users expend few 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 execute the instrumentation. Users that participate in this program will be compensated with reduced rate or free software.

7. All operational data collected from users is delivered exclusively to the DCDL. The raw information does not go to the publisher. DCDLs agree to legal confidentiality between themselves and users. Once the raw information is scrubbed and user identities are not traceable from the composite profile, the profiles are passed back to publishers. This allows publishers to hone their products to the preferences of their user base.
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- β versions that do not have profile collection instrumentation.
9. The number of participant users is determined by the DCDL. Fewer users almost certainly means that it will take longer to collect enough data. Also, the diversity of participant users must be taken into consideration by the DCDL. After all, the DCDL is responsible for ensuring that their sample population mimicks the general user population.
10. Publishers compensate DCDLs for their services. The DCDL owns all collected data. Because the data has economic value, fees placed on publishers by the DCDL are reasonable. We envision publishers subscribing to the DCDL and receiving only the information they want on a per version basis.
11. The DCDL provides the publisher with a tool that automatically embeds instrumentation into a product. This tool provides a publisher with a variety of options, each of which collect different types of user information.
12. Because these methods are automated, this approach is *repeatable* and *reproducible* (meaning that different DCDLs would produce the same composite information to the publisher provided they received the same raw data from the field).

4 Related Approaches

While our proposed model for combining DCDLs and instrumentation is quite unique, it admittedly leverages previous approaches. More specifically, it is conceptually similar to Netscape 4.5's Quality Feedback Agent and the product known as PureVision. In this section, we will discuss these and other related ideas.

PureVision was a software product offered by Pure Software, released in 1995, but no longer available today. The product performed crude monitoring functions. It worked in a manner similar to our model: 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 information back to the publisher.

Our second example is Netscape 4.5, which 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 was not well received.

In contrast, Microsoft has had a far better relationship with users in its official beta test programs. 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. Also, Cusamano and Selby have alleged that Microsoft gathers detailed user profiles from specifically instrumented versions of their software [2].

So what can we surmise from these anecdotes? First, we see that companies really want to know what is going on with their products at user sites. And secondly, we see that there are users like Microsoft's β -users that are very willing to participate to help a publisher if they receive something in return.

What still remains to be seen, however, is whether users will have more trust in a DCDL than they do in publishers. That is, is a DCDL an acceptable intermediary between users and publishers? As long as the confidentiality agreements between: (1) the DCDL and publisher, and (2) DCDL and user are binding, we believe users will. And we can further strengthen the trust between the user and the DCDL by performing some degree of data scrubbing at the user site, provided that the DCDL remains confident that incoming scrubbed information is legitimate and not a forgery. And in fact, doing so may increase the number of users that would be willing to participate, thus increasing and accelerating the amount of field data collected.

5 Key Benefits of Our Model

While there are numerous reasons for creating accurate operational profiles for mass-marketed software, we believe that the three main ones are:

1. Accurate operational profiles provide publishers with the need, hopefully, for fewer “point” releases if the publishers properly employ the data during in-house testing. Why? Because they will be able to test and design according to the desires of their customer base. Thus, reliability goals should be achievable in earlier versions.
2. Accurate operational profiles provide yet another benefit to publishers: the ability to decrease version “bloat” over time. That is, the information that is fed back to publishers will include feature usage profiling. When features are found to be ignored by most users, publishers can have the ability to optimize their products in a manner that best mirrors the interests of their users.
3. Accurate operational profiles provide users with the opportunity to have a greater say in which direction a product evolves.
4. Accurate assessments of feature usage could be very useful for Application Service Providers (ASPs). ASP businesses charge users a fee based on the extent to which software is used. You can think of the ASP business model much like a toll road. The more you drive on the toll road, the more you pay. So for example, instead of buying a COTS package with 1,000 features of which user X might only wish to use 5, the ASP could charge X for each use of those 5 features. Our model could be used by the ASP as a way to bill users based on which features they use (some features might cost more) and how often they use those features.
5. Different types of profiles can be created for different types of persons within the publishing organization. So for example, the marketing and advertising department may want a completely different type of information than the developers. By simply modifying the instrumentation and what information is collected, many different groups within the publishing company can be satisfied.

6 Summary

Accurate operational profiles provide publishers with a way to better tailor their offerings to their users. Unfortunately, for most mass-marketed products this information is unknown, even though these products have thousands of users.

Operational profiles have other important applications. They are needed for reliability prediction and metrics for when to halt testing. And they will become even more important as component-based integration becomes the standard development paradigm (in order to assess *interoperability*).

We admit that capturing operational profile information *after* a product is released is less desirable than having the information *prior* to the release of the first version of a product. But because this is not feasible, we contend that by immediately applying this process to the earliest version possible, we can greatly reduce the downstream costs of future versions by aiding the publisher to better tailor their product to the nuances of their user base.

We envision that our proposed approach will initially be applied to complete applications (*e.g.*, Microsoft Word). Once it has been applied at that level of granularity and we have more experience, we believe that the model can also be applied to smaller sized objects (*e.g.*, plug-ins and components). This clearly has the potential to decrease the cost of software development by fostering component-based software engineering.

And finally, we view this process as the first step toward limited software warranties and certificates of quality. Once we can gather accurate information about how a product is used, that opens the door to then collect information about how well the product is behaving. Once armed with both pieces of information, express warranties can be provided that define the level of quality that “a typical and reasonable” user can expect [5]. Such a warranty could read as follows:

When feature Z of version Y of product X is used on machine M configured in manner C , the product will fail approximately one time per thousand executions of Z given that only even integers are fed into Z .

However until that day arrives, a fact sheet can suffice. For example, having a statement such as the following, although not a warranty, still provides insight to a potential user as to the reliability that they can expect and under what assumptions they can expect it:

For version Y of product X , when feature Z was tested one million times using even integers on a machine M configured in manner C , X crashed 1000 times.

The implication from such an oath is straightforward: if someone else plans to use Z (of X and Y) in a similar manner (*i.e.*, even integers) on the same type of a machine and configuration, then they can expect a similar percentage of failures. To make this believable, however, detailed operational information concerning C and M must be published to limit ambiguity. Why? Because the journey toward publically exposing product defectiveness via limited warranties and fact sheets cannot begin until we have accurate operational profile information from which certificates proclaiming quality (or a lack thereof) are tied.

7 Future Work

This paper has focused on providing information back to publishers on how *their* products are used. We assumed that source code was instrumented to detect feature usage and scripts were used to profile the environment. If we were to instead instrument software binaries without assuming cooperation from the publisher, then it would be possible to also license

the results to competitors. We plan to look into the technical feasibility of doing so in the near future.

References

- [1] L. BINGLEY. PureVision: Shedding Light on Black Art Betas, APT Data Services, Inc, New York, June 1995, page 17.
- [2] C. CUSAMANO AND R. SELBY. *Microsoft Secrets*. The Free Press, 1995.
- [3] J. D. MUSA. *Software Reliability Engineering*. Wiley, 1998.
- [4] J. D. MUSA, A. IANNINO, AND K. OKUMOTO. *Software Reliability Measurement Prediction Application*. McGraw-Hill, 1987. ISBN 0-07-044093-X.
- [5] J. VOAS. User Participation Based Software Certification. *IEEE Computer*, To appear in 2000.
- [6] J. WHITTAKER AND J. VOAS. Toward a More Reliable Theory of Software Reliability. *IEEE Software*. Submitted.