

Mitigating the Potential for Damage Caused by COTS and Third-Party Software Failures

Jeffrey Voas

Reliable Software Technologies

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 years ago. Today’s software systems are complex “systems of systems”, and developers must accept the fact that substantial portions of these composite systems will be provided by other developers. Losing control over *every* aspect of a system’s functionality may worry the parties that are legally liable for the quality of the complete system. Those parties need assurance that each component will tolerate each other.¹

Software reuse has the potential to massively increase the rate at which information systems are built while reducing the costs of building these systems. Software reuse generally occurs in one of two ways: (1) purchasing Commercial-Off-The-Shelf (COTS) “generic” software, and (2) reusing one’s own software modules from project to project through shared libraries.² But each of these methods run the risk that the complete system will suffer from problems caused by the reused or acquired software. This paper presents a methodology for predicting whether this is likely to occur (as well as presenting approaches for reducing this likelihood).

Software reuse is easily justified. The world’s software and services market is rumored to be a 300 billion US dollar industry per year. Software design and development costs could be significantly reduced with increased code reuse. Even the best programmers only churn out 10 lines of code per day, which for systems such as cellular phones (that now have around 300,000 lines of code in them), have made custom software development very expensive [1]. If, for example, 100,000 lines of code could be reused, 10,000 programmer-days of effort could be saved.

¹When two distinct software components interact correctly, this is referred to as *non-interference* between the components.

²There is a special case to (1) and that is procuring custom software functionality from a third-party vendor. Here, the software may or may not be new, but it presents the same risks of reused software from a quality standpoint, and thus the methodology that we will propose here will apply to this situation.

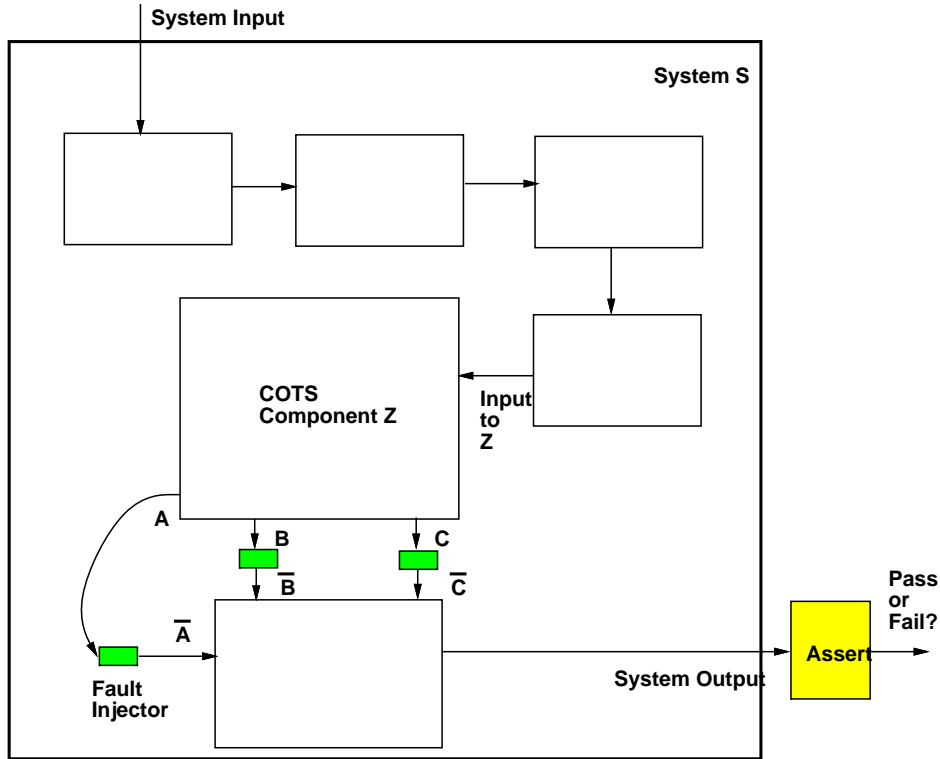


Figure 1: Typical System S that includes a COTS Component, Z .

Reductions in development costs and decreases in time-to-market encourages reuse. But reuse is not a panacea. The Ariane 5 disaster demonstrated that fact [2].

Reuse enables bigger and more complex software systems. 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. Consider that even today’s best systems still have software defect densities of around 3-6 faults per KSLOC (if you count every fault, no matter how small it may be) [4]. Interestingly, that rate has held constant for the last two decades regardless of the shift to object-oriented technology, software reuse, automated debuggers, better test tools and compilers, stronger type safety, etc. Because all of the new ideas have not decreased this rate, it would appear that defect densities of 3-6 faults per KSLOC is a “natural phenomenon.” The combination of this fixed defect density and larger systems suggests that the reported crisis (See W. Gibb’s article, “Software’s Chronic Crisis”, Scientific American, 1994) is very real.

If we are unable to overcome this naturally occurring phenomenon, then we must get better at diminishing the consequences of faults. The methodology proposed here, which mitigates the potential of dangerous program states occurring during software execution,

is one step in that direction. As it turns out, this methodology is particularly amenable to generic software components that are expected to operate in distinct environments or throughout a family of products.

2 Methodology

Our methodology is based on the premise that defect-free software is so rare that it is fair to consider it as an oxymoron. Further, we dismiss the idea that component developers will ever guarantee absolute correctness of components. Even if they were to via proofs-of-correctness, there is no guarantee that the component will not interfere with the remainder of the system, causing the system to have lower quality.³

Thus we assume that each software component will have failure modes. The goal of our methodology is to determine the severity of the failure modes, and only thwart those modes that cause intolerable system problems. By only going after failure modes that lead to unacceptable consequences, quality improvement resources are wisely conserved. We cannot expect to thwart every failure mode, and if we were to try, we would end up having done a poorer job of thwarting the failure modes that are the more worrisome ones.

Figure 1 illustrates an example system (we will return to this example throughout the remainder of this article). Figure 1 shows an information system, S , that contains a COTS component Z . Z has three output variables: \mathbf{A} , \mathbf{B} , and \mathbf{C} . To determine which failure modes from Z cannot be tolerated by S , we will use fault injection [3, 5] to forcefully corrupt (modify) the information contained in \mathbf{A} , \mathbf{B} , \mathbf{C} , or some combination of these. To keep this example simple through the paper, we assume that only \mathbf{A} is forcefully corrupted via fault injection. In Figure 1, we denote the altered information in \mathbf{A} as $\bar{\mathbf{A}}$.

$\bar{\mathbf{A}}$ is not produced naturally by Z (for the input vector that Z was given). $\bar{\mathbf{A}}$ is artificially corrupted information exiting Z that is introduced into S by a fault injection algorithm. Naturally occurring corrupt information exiting Z may or may not cause S to produce undesirable output. Fault injection allows us to observe how artificially corrupt information propagates (as well as the consequences it causes). This provides a nice basis from which to reason about how actual corrupt information will affect S .

As shown in Figure 1, undesirable outputs exiting S are detected by an assertion that monitors S 's output during fault injection analysis. This assertion defines what outputs are acceptable. These assertions are derived from S 's specification or requirements.

By corrupting the information stored in \mathbf{A} , we discover corrupt output states from Z

³Composing two correct components together does not guarantee a correct composite. This is due to the possibility of *interference* and non-functional behavior.

that cause S 's assertion to fire.⁴ Assertions fire when unacceptable outputs are produced. Unacceptable outputs can be those that are unsafe, vulnerable, incorrect, etc.

By artificially corrupting the results returned by a component, we stand to find component outputs that we do not want produced. The key question then becomes: *are there legal component inputs that may be selected that can cause the component to naturally produce those outputs that fault injection has determined are intolerable to the system?* If the answer is “no”, then we have confidence that the component is well-suited for the system. If the answer is “yes”, there are tasks that need to be performed before Z is incorporated into S . This paper suggests techniques that can answer this question as well as techniques for making S tolerate Z if the answer is “yes.”

3 Mitigation Strategies for Z

From this point forward, we will assume that fault injection uncovered corrupted states that fired S 's assertion. We will now suggest mitigation strategies that address the above question. At this point, it is premature to abandon Z (meaning to not embed it in S), but we should be cautious. We do not yet have proof that Z is capable of behaving in a manner that is identical to those behaviors that were employed during fault injection (that resulted in unacceptable system outcomes), but likewise, we have no guarantee that the component will not.

To determine whether there are input vectors to the component that can naturally produce those \bar{A} s, we propose a set of generally well-known techniques. To our knowledge, however, this paper presents the first application of these techniques to mitigating this COTS problem. There will be cases, however, where these well-known techniques will fail to answer our question because of undecidability problems.

Our recommendations here place the onus on the developer to demonstrate the integrity of his or her components. After all, it is the developers that will profit if the components are adopted. If the developer fails to provide this assurance, there is still one other mitigation strategy that we will recommend that can be performed by the party that is considering adopting the component. This final strategy does not require cooperation from the component's developer.

Our complete methodology is shown in Figure 2. The methodology involves performing fault injection first, and then performing one or more of the mitigation processes. For the developer of a suspicious component, static fault tree analysis, backward static slicing (with testing), and wrapping are three alternatives that enable the developer to demonstrate

⁴It is unlikely that we will find every Z from limited fault injection, but even if only a few are found, information about them can enable us to add greater robustness to S .

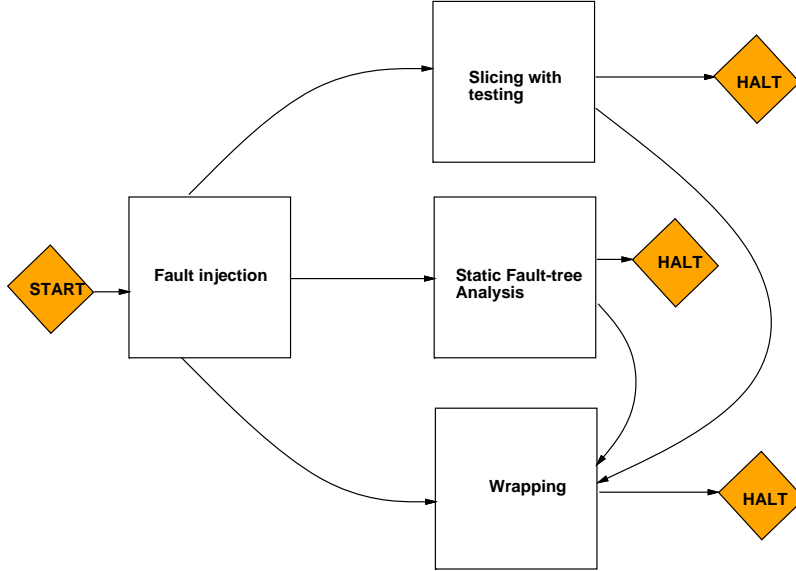


Figure 2: Mitigation Process.

component integrity:

1. Static Fault Tree Analysis (SFTA) can prove (by contradiction) that certain failure classes are not possible from Z , even if Z were to receive bad input data from the remainder of the system.
2. Backward static slicing from those variables that were corrupted by fault injection (and caused S 's assertion to fire) can establish those slices that need concentrated testing. This provides an *argument* that those failure modes in $\bar{\mathbf{A}}$ are not possible from Z , but this is not a *proof*.
3. Wrap Z to either filter Z 's inputs, outputs, or both. (Since all we have at this point are $\bar{\mathbf{A}}$ s, wrapping outputs makes more sense.)

Developers need to perform only one of these processes to provide the necessary assurance. Component developers are the best candidates for performing the first two analyses because they have access to the internals of the component. The adopter for the component will likely not have that information. If the developer is unwilling, wrapping, the third alternative in our list, can be performed by the system integrator (without help from the developer and without access to the code in Z).

We will now walk through each of these mitigation strategies and how they address our key question.

3.1 Static Fault-Tree Analysis on the Component

Static fault-tree analysis is a design technique that was developed by Watson in the 1960's. Static fault-tree analysis assesses the causal relationship between events in a process. Software fault-tree analysis is simply the application of static fault-tree analysis to software.

Safety analysis techniques generally attempt to demonstrate safety by using a “backwards” approach; catastrophic output events (hazards) are determined first, and then the designer works backwards from the hazards to events that will happen earlier in time to either show that: (1) hazards cannot occur, or (2) the probability that they could occur is low. For fault-tree analysis to be practical, there must be a small set of hazards. If not, the time involved in performing the backwards analysis will become intractable.

Fault-trees are graphical representations of events; the graph is shaped like a tree data structure. The root node, or what is termed the “top node”, represents an undesirable event that hopefully cannot occur. The other nodes of the tree represent parallel and sequential events that potentially could cause the root node event to occur. These other events are logically connected using OR-gates and AND-gates. For a given event node x , its children are the necessary pre-conditions that are believed might cause x to occur. During fault-tree mitigation, trees are continually expanded by creating subtrees until leaf-events are created for which we can assign a probability for the leaf-event or leaf-events are created that cannot be decomposed into subevents.

To apply fault-tree analysis to our problem, those \bar{A} s that caused S 's assertion to fire will be OR-ed together as the top event. The underlying events will represent: (1) failures of subcomponents of Z , (2) failures of external entities upon which Z depends for input information (such as a database or a call to a system library function), and (3) failures caused by a combination of the previous two failure types.

To prove that these types of failures cannot produce those corrupt outputs from Z that fired S 's assertion, proof-by-contradiction is employed. If, during these attempted proofs, a branch exists that does not result in a logical contradiction, then the analysis has uncovered events that can lead to those undesirable \bar{A} s. This means that there *are* harmful outputs from Z that must be protected against. (Section 3.3 presents an approach for doing this.)

3.2 Slicing and Testing the Component

Besides fault-tree analysis, the developer has the option to perform slicing and then perform slice testing. Slicing and fault-tree analysis are both techniques that perform *impact analysis*, but the processes that they employ to study casual relationships are quite different. Slicing and slice testing, unlike fault-tree analysis, does not prove that the output events are impossible. The results from slicing and slice testing are empirical. Even though these approaches to impact analysis are different, the results from these different techniques can

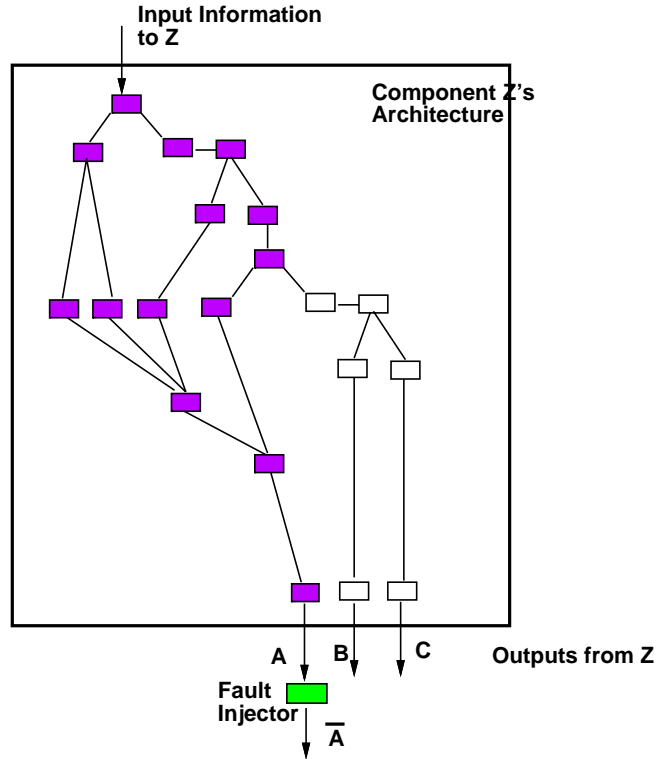


Figure 3: Internal subcomponents of Z that affect A .

both demonstrate that certain output events are unlikely. And this information is precisely what is needed for mitigation.

Figure 3 hypothetically illustrates the internal subcomponents of Z . The darkened subcomponents combine to create what is called a “static slice” of the program for variable A . A backwards static slice from A shows those internal computations that can potentially affect the value in A . Depending on the level of granularity of a slice, the darkened subcomponents could represent statements, blocks, modules, etc.

After the static slices are found, the slices must be thoroughly tested to determine whether those \bar{A} s can be output from Z .⁵ This is what we previously referred to as *slice testing*. Note that generating test cases to exercise a slice can be a difficult task, and like almost all problems that involve test case generation for code coverage, it can suffer from undecidability problems.

In addition to testing the slices using “normal” inputs to Z , like fault-tree analysis, it is prudent to also test in a manner that simulates failures of external components upon which Z depends for input. Thus, we should test the slices using corrupted inputs to Z to see if that

⁵It is out of the scope of this paper to define “thorough”, but as a minimal condition, all of the code in the slice should be exercised.

corrupt input information forces Z to produce those $\bar{\mathbf{A}}$ s. This testing can be accomplished by hooking a fault injection tool to the test case generator. The fault injector intercepts the test cases and corrupts them before they are handed to the component. Note that this alteration of the test case may force subcomponents associated with different slices than the one in question to be exercised. If this occurs, then we immediately get a result showing that corrupt information to the component cannot cause the undesirable outputs of concern (affiliated with the slice in question).

In summary, if those $\bar{\mathbf{A}}$ s do not occur during slice testing, confidence in Z is increased. If those $\bar{\mathbf{A}}$ s do occur, wrapping of Z should be considered. (See Section 3.3).

3.3 Wrapping the Component

Whether a particular $\bar{\mathbf{A}}$ fires S 's assertion is dependent on the input state of S . Thus the same $\bar{\mathbf{A}}$ may fire S 's assertion when S is one state and not fire S 's assertion for a different state. For this reason, what is defined as an unacceptable output from a component may not always be mitigatable by the first two approaches. But with wrappers, a test of the system state can be included before an output event from a component is thwarted by the wrapper.

If the developer does not provide convincing evidence that Z cannot return those $\bar{\mathbf{A}}$ s using static fault-tree analysis or slicing, software wrappers are the next alternative. Or if the developer uses static fault-tree analysis and slicing and observes that those undesirable $\bar{\mathbf{A}}$ s are possible, then wrapping will be a means for making those undesirable $\bar{\mathbf{A}}$ s impossible.

Usually, a software *wrapper* is a software encasing that sits around the component and limits what the component can do with respect to its environment. (There is also another type of wrapper that limits what the environment can do to the component.) Wrappers are tasked with detecting that undesirable outputs are going to occur and then keeping them from occurring. Wrappers act similarly to post-conditions (which test to see that certain truths hold). To keep undesirable outputs from occurring, software wrappers have the ability to modify a component's output. Wrappers do not modify a component's source-code in an invasive manner, (*e.g.*, removing lines of code from the component), but instead indirectly modify the functionality of the component.

To perform wrapping, two different wrapper approaches can be taken. One approach ignores certain input vectors to the component and thus does not allow the component to execute on those inputs. That limits the component's functionality and will be useful if fault-tree analysis shows that failure types 2 or 3 are possible. The other approach captures the output before the component releases it, checking to ensure that certain constraints are satisfied, and then only releases outputs from the component that satisfy the wrapper's constraints.

In our methodology, fault injection initially reveals which outputs from Z lead to system-level problems. To build component wrappers from those $\bar{\mathbf{A}}$ s, several options are available.

First, tables can be built that contain precise information about undesirable corrupted states (along with the system states for which they are undesirable). If these states are observed as the program runs, the wrapper can halt execution. A second option is to use artificial intelligence approaches to build heuristics that classify undesirable outputs, and if outputs that satisfy the heuristics are observed, they too can be thwarted. Currently, we are building several different prototypes that use Connectionist learning to test how successful these learning techniques are as wrappers.

In summary, wrappers are a clever way to build protection from components. But wrappers are only as good as those persons who build them. And if fault injection is used as a basis for building wrappers, then the wrappers will only be as good as the fault injection analysis which determined what events required thwarting.

Once wrappers are built and attached to the component, black-box testing of the component with the wrapper should be reapplied to ensure that those $\bar{\mathbf{A}}$ s that triggered S 's assertion are no longer possible. Also, fault injection can be applied to the input vector of the wrapped to component to test the strength of the component's wrapper.

3.4 Genetic Algorithms

Although it is premature to advocate *genetic algorithms* as a mitigation strategy for this problem, we should mention that genetic algorithms, which are widely being researched as methods for automatic test-case generation, may someday play a useful role here as another means for mitigation. Given those $\bar{\mathbf{A}}$ s that were unacceptable, and given that we know what \mathbf{A} s were produced by the inputs (prior to fault injection being applied), we may be able to train genetic algorithms to search the input space of Z and see if it can find input vectors (test cases) that will create those $\bar{\mathbf{A}}$ s.

We have already developed a test-case generation tool that uses genetic algorithms to find test cases that exercise various parts of the code. To reach these parts, certain constraints must be true in the state of the executing software. In terms of fault injection, we know what $\bar{\mathbf{A}}$ s (which are really nothing more than system states) are undesirable to S . The question then becomes whether there are input vectors to Z that can create them. Thus we believe that it may be possible to apply our genetic algorithm-based test case generator to this COTS problem.

We are currently experimenting with this idea in other research related to information security. There, we have connected a fault injection tool that attempts to break into systems to our genetic algorithms. The reason that we are doing this is to see if the genetic algorithms can find exploits that could be used to break into systems in similar ways to how the fault injector was able to break into systems. (The fault injector that we are using has been fine-tuned to discover corrupted program states that will allow systems to be broken into.)

If successful, this security-related research initiative will have found a way to move information security away from a penetrate-and-patch paradigm to a “patch-before-penetrate” paradigm. Further, if we succeed in linking fault injection to genetic algorithms, there are an entire range of problems in testing, safety, and security that can be addressed (which today are still awaiting solutions).⁶ But at this point, our efforts here are in the early stages. If successful, however, this approach will be applicable as a fourth way to mitigate this COTS problem.

4 Conclusions

We are taught as children to not accept things from strangers. But to get software systems to market more quickly, we must accept software written by strangers. Today, the rumor that an anticipated product will be delayed can plummet the value of a company’s stock by hundreds of millions of dollars. And such rumors will increase the morale of a company’s competitors.

It is still very difficult to produce highly reliable *custom* software when working from a custom specification, even after 30+ years of experience as an industry. Given this, it is implausible to believe that we are yet in a position to expect that *generic* components will fit more reliably into a custom system than customized components. For this reason, we have recommended component assessment processes that ensure component compatibility on a system-by-system basis. These processes test the integrity of components with respect to the environment in which they need to be trustworthy. These tests increase confidence that the components will integrate seamlessly into a system with less interference and fewer side-effects.

It has been predicted that software component catalogues are “the future.” These catalogues will allow consumers to simply read through a listing of components, select those of interest, and plug them in. Clearly this is futuristic thinking, but how far away we are from it is unclear. It could be years or decades away.

Component catalogues are instrumental in designing hardware systems. In electrical engineering, designers first discover what components are available, and then they design the remaining system around those parts. In our opinion, if component-based software is to evolve into a catalogue industry, it will begin by component consumers demanding specific warranties about specific component behaviors from component suppliers. This paper has presented fault injection as a means by which component adopters can determine what warranties they need to request, and we have also provided suggestions for how developers

⁶For example, wouldn’t it be great to know about members of the input space for a flight-control system that will cause the plane to crash if they are ever selected.

can provide these warranties (using fault-tree analysis, slicing, or wrappers). As components mature via usage in more and more unique environments, these warranties can become more generic, allowing for a more “off-the-shelf” (as is) warranty to be offered with a specific component. From here, catalogues with these components can evolve.

Admittedly, few component developers will cheerfully perform our recommended mitigation analyses. It will likely fall back onto the shoulders of the adopter to perform tasks such as wrapping if the developers refuse. However, if system integrators were to universally demand such actions, component developers would have greater incentives to certify the quality of their components.

Acknowledgements

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, 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] S. BAKER, G. McWILLIAMS, AND M. KRIPALANI. “Forget the Huddled Masses: Send Nerds”, *Business Week*, July 11, 1997.
- [2] Prof. J. L. LIONS. Ariane 5 flight 501 failure: Report of the inquiry board. Paris, July 19, 1996, available at http://www.cnes.fr/actualities/news/rapport_501.html.
- [3] J. VOAS AND G. MCGRAW. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley and Sons, New York, 1997.
- [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, G. MCGRAW, L. KASSAB AND L. VOAS. A Crystal Ball for Software Liability. *IEEE Computer*, 30(6):29–36, June 1997.