

# Reducing Uncertainty About Survivability

Jeffrey M. Voas, Gary E. McGraw, and Anup K. Ghosh

Reliable Software Technologies Corporation

Suite 250, 21515 Ridgetop Circle

Sterling, VA 20166 USA

Fax: 703.404.9295

Phone: 703.404.9293

{jmvoas,gem,agghosh}@rstcorp.com

<http://www.rstcorp.com>

## 1 What is Survivability?

Survivability is not an all or nothing phenomenon. There are varying degrees of surviving. The question is, can survivability be assessed? The need to measure survivability is not unlike the need to measure risk, quality, safety, and so on. Without some way to know how close we are to a goal, it is difficult to know whether any development steps we are taking are hindering or helping a system.

Truly knowing how survivable a system is requires two things: 1) a precise definition of what survivability is, and 2) a precise way to measure progress at any intermediate developmental point. A workable definition for “survivability” has alluded us to date. In fact, survivability has remained a muddled mixture of other ideas including: reliability, fault-tolerance, safety, availability, etc. That means from a measurement standpoint, survivability is no easier to quantify than any of the other characteristics listed above — probably less so, since survivability is a composite of the already ill-defined other things.

Ultimately, survivability is a measure of strength against attack combined with an analysis of logical defectiveness. Along these lines, the last two decades of computer science research have focused on: 1) ways of developing logically correct software (formal methods), and 2) testing as a means for demonstrating correctness. But today’s distributed systems preclude both of these approaches since real systems either have few (if any) specified formal properties and are generally untestable on top of that. These deficits are a direct result of the complexity and size of today’s systems.

In the final analysis, developing and demonstrating survivable distributed systems remains an important and currently unattained research goal. The problem has never been sufficiently solved. The lack of a solution leaves an enormous burden on today’s users who need confidence that their electronic transactions are private, safe, and secure. This is especially relevant considered in light of the upsurge in Internet usage.

The most prevalent approach to conquering complexity is the “divide and conquer” approach. The idea is to decompose a large problem into subproblems, and solve the smaller

problems independently. Once smaller problems are solved, the sub-solutions can then be integrated into a global solution. Distributed object systems are built using a closely-related design paradigm.

The problem with assessing the quality or strength of distributed systems stems both from the interactions between the subcomponents as well as from the many different entry points into the system. We can test one component in isolation until the sun burns out, but such testing will tell us little about what happens when that component is made part of a unified system.

It is our contention that the only tractable solutions for the complexity we are faced with — complexity that will only increase with time — is to assess survivability at a macroscopic, not microscopic, level. We believe that survivability can only be improved if deficiencies are measured at a high system level. Testing is used (and should still be used) to identify bugs at lower, sub-system levels. Thus system-level analysis remains unsolved. However, methods for predicting the impact of failures and attacks against subcomponents at the sub-system level can assess the survivability of large-scale distributed systems. Testing methods for developing more survivable components can be enhanced to some degree by formal methods; but methods for assessing system survivability must employ some notion of simulated failures. Fault injection approaches are best suited for this goal.

This paper discusses how fault injection can be applied at the whole-system level. Most people think of fault injection as a lower-level approach that requires source code. But fault injection works equally well at higher levels of abstraction. In fact, application of fault injection at higher levels helps to ferret out those subcomponents causing the greatest system-level survivability risks. Fault injection simulates failures. Given simulated failures defined at a high enough level, we can assess system survivability. Since this assessment is made with respect to the remainder of the system, it has the potential ultimately to provide a much needed overall view of the entire system.

## 2 Composing Survivability

It has been conjectured that building software systems from components can potentially increase the reliability of a system relative to coding the same system from scratch as a single, monolithic entity. The rationale for this claim is that: (1) more time spent on design and analysis up-front in project development will improve system reliability and (2) building systems from tried and tested components will result in more reliable systems. The basis for this argument originates from principles in software engineering and practical experience rather than scientific analysis.

For mission-critical, enterprise-critical, and safety-critical systems, good engineering principles by themselves are insufficient for assuring survivability. Some mechanism for assessing the survivability of a computation-critical application must be applied in order to provide the required assurance before application deployment. In other words, while good software engineering principles may be necessary in order to build robust systems, they are not a guarantor of survivability. In fact, some design approaches such as *diversity*, which has not been shown to be useful for creating more reliable software systems, may be just what is required to build survivable systems.

In order to highlight the problem of assessing survivability of distributed object systems, we first present the problem of composing system reliability through analytical methods. In composable/analytical methods, systems of components are represented as series and parallel constructions. In a series construction of physical components, the failure of one component will result in the failure of the system. The reliability of such a system can be composed in a straightforward way from the individual reliabilities of the component parts. If the failures of component parts are assumed to be independent, an equation for the reliability,  $R$ , of the system can be written as a function of time [3]:

$$R_{series}(t) = R_1(t)R_2(t) \cdots R_n(t) = \prod_{i=1}^n R_i. \quad (1)$$

In order to increase the reliability of systems, parallel construction of components can be used to negate the consequences of a failed component. In a parallel construction of systems, if one component fails, another parallel component can cover the functionality of the failed component. In order for a parallel system to fail, all redundant parallel components of the system must fail. From a probabilistic view, the unreliability of the parallel system,  $Q_{parallel}$ , is equal to the product of the unreliabilities of its components assuming that the failures of components are independent. In equation form, the unreliability of the parallel system can be represented as [3]:

$$Q_{parallel}(t) = Q_1(t)Q_2(t) \cdots Q_n(t) = \prod_{i=1}^n Q_i. \quad (2)$$

Since the sum of the reliability and unreliability must equal 1.0—formally  $R(t) = 1.0 - Q(t)$ —we can derive an equation for the reliability of parallel systems [3]:

$$R_{parallel}(t) = 1.0 - \prod_{i=1}^n (1.0 - R_i(t)). \quad (3)$$

Many systems can be represented as combinations of parallel and series constructions. Reliability block diagrams can be used to visualize the series/parallel constructions and to reduce these constructions to analytical formulae.

Like physical systems, software components can be combined in series and parallel constructions. Most often, components are interconnected in series in order to perform functions demanded by the application. That is, component  $A$  may make a request to component  $B$ , which may in turn make a request to component  $C$ . Software components can also be constructed in parallel. Parallel components perform the same functions, process identical inputs and produce outputs that if correct should be identical.

The reliability of software systems, however, is difficult to determine analytically for several reasons. First, unlike mechanical systems, the reliability of a series construction is not necessarily less than or equal to the reliability of the weakest component. In software, fault masking can occur in which a failure in one component can negate the effect of a failure in another component. Thus the reliability of the system could potentially be higher than the product of the individual component reliabilities. In parallel software component systems, common mode failures often violate the validity of assuming independent failures.

Common mode failures can result from errors or ambiguities in the specification, or errors in coding in two or more components that result in identically incorrect outputs. The upshot of a common mode failure in a parallel system is that the increase in reliability expected from redundant parallel components is not realized.

Beyond the problems with analytical composition of system reliability for software systems, perhaps a more pressing practical issue is that there is currently no consistent and accurate way to estimate the reliability of a software component. Unlike physical components whose failure rates are based on physical wear and tear, software component failures are due to flaws in design and coding. Software flaws are only manifested when an input causes the software flaw to infect a data state after which the infected data state propagates to a discernible output [7]. The failure of some component, then, is not only dependent on the existence of a flaw, but also on the inputs the component receives. If the software component never receives an input which exposes any flaws, an external observer may conclude the software is 100% reliable. On the other hand, if an input which exposes a flaw is received in every execution, then the reliability may be characterized by an external observer as 0%. What this anecdote reveals is that the software component's true reliability cannot be accurately estimated since any reliability estimate will depend not only on the component code, but also on the environmental conditions of the component, *i.e.* the input it receives.

See Singpurwalla for a discussion regarding software failure rates [5].

### 3 Using COTS Components

The discussion of the preceding section highlights the problems in using analytical methods to assessing the reliability of component-based software systems. These same difficulties apply equally to composing system level survivability. A critical concept in evaluating the survivability of a software system is that the survivability measure is context-dependent. System survivability is dynamic and is a function of a number of factors normally excluded from analytical models: the environment in which components are executing, the uses to which the software is subjected, the reliability of connections between components, and the load which the system must handle.

The component-based model of large-scale software systems can also constrain system-level evaluation to approaches that are source-code independent. In many distributed object systems, the source-code for an object is not available for analysis. One of the fundamental concepts in component-based software systems, is that a client does not ever need to know the implementation details of a request-handling object. The implementation details, or in other words the source code, is completely encapsulated by the interface to the object. It will often be the case in large-scale application development that a software developer will be building an application that consists of some components that are written in-house and others that are Commercial Off The Shelf (COTS). The problem the developer or system analyst faces is how to analyze the robustness of an application that is built from some combination of locally-developed and COTS components. Lastly, many distributed object systems are now being employed to build next generation systems composed of legacy components [2]. Often times the legacy source has not been documented or maintained, and it may even be written in out-dated languages. These conditions render source code analysis difficult at best

and sometimes just plain infeasible. These examples point to the dire need in commercial applications for methodologies and tools to evaluate the survivability of systems composed of COTS and legacy components.

To assess the survivability of systems potentially composed of custom-developed, legacy, and COTS components, the methods we advocate examine the resilience of the interfaces between components [6]. Interfaces are the mechanisms for passing information between components. All inputs and outputs of a component define an interface, with the final interface being the mechanism by which outputs are passed to the end-user. Inter-component interfaces are the glue that binds components and are often the weakest links in a complex system.

The interface a designer creates for a component reflects the assumptions the component designer makes about the external environment. Distributed object systems can reduce the complexity of creating large applications by componentization. Componentization is the process of dividing a complex problem into manageable parts, each representing a different service that is provided to the system. Componentization can make design, development, and testing simpler by making possible the application of resources on smaller partitions of the problem that can be more easily grasped by component developers. However, while componentization can reduce the complexity of a monolithic system, it correspondingly increases the complexity of interfaces. This relationship is summed up well by Leveson on page 36 of [4]:

One way to deal with complexity is to break the complex object into pieces or modules. For very large programs, separating the program into modules can reduce individual component complexity. However, the large number of interfaces created introduce uncontrollable complexity into the design: the more small components there are, the more complex the interface becomes. Errors occur because the human mind is unable to comprehend the many conditions that can arise through interactions of these components. An interface between two programs is comprised of all the assumptions that the programs make about each other... When changes are made, the entire structure collapses.

The quote illustrates the trade-off in componentization. On the one hand, componentization can alleviate the difficulties in large system design by partitioning system functions into components. Components can be individually designed, developed, and analyzed for dependability. This process is much more manageable than designing, developing, and analyzing a single monolithic application source. On the other hand, the complexity of interfaces grows when partitioning a system into its component parts. Provided that there is some assurance that the individual components are trustworthy and resilient, the weak link in such a system may well be the interfaces or connections between the components. Distributing system functions across networks, supporting heterogeneous platform implementations, and re-using commercial components for which the implementation is unknown, can introduce anomalies in the execution of the application. As mentioned previously, the dependability of a large-scale distributed system is not easily composed from its component parts.

## 4 Assessing Survivability

In light of the problems generating analytical solutions, the constraints of systems composed of COTS components, and the critical nature of component interfaces, the methodology we advocate simulates failures in components by corrupting component interfaces. Perturbing component interfaces with anomalous events under simulated stress provides valuable information for how the system will react under real stress, *i.e.* after system release [6]. The survivability of the distributed system can be thought of as a prediction of the ability of the system to tolerate component failures resulting from malicious and non-malicious anomalies within components and from external sources.

The assessment of distributed object system survivability can be partitioned into two types of analysis: (1) interface propagation analysis and (2) survivability analysis. Interface propagation analysis uses fault perturbation functions to inject anomalous events into component interfaces and to observe how the corruption propagates across components [6]. Survivability analysis dynamically evaluates survivability assertions about a system subjected to fault perturbation in order to determine if the survivability assertions have been violated. Both analyses can be performed concurrently during dynamic execution analysis.

### 4.1 Interface propagation analysis

The goal of interface propagation analysis (IPA) is to determine to what extent the failure of one component will corrupt other components in a system—and ultimately the entire system itself. Interface propagation analysis is a method for determining whether the failure of component  $B$  will cause the failure of component  $A$  and subsequently the failure of the composite system. Questions that IPA can be used to address are: (1) will corrupted inputs to  $B$  result in corrupted outputs from  $B$ ? (2) will corrupted outputs from  $B$  result in a corrupted input to  $A$ ? and (3) will corrupted outputs from  $B$  corrupt subsequent outputs from  $A$ ? Interface propagation analysis can be used at the component level and at the system level. At the component level, interface propagation analysis provides a prediction of how likely it is that corrupted data on the input interface to a component will cause corrupted data on the output of the component. This information also provides a measure of the testability of such a component. That is, if a corrupted input almost always results in a corrupted output, the component will be observed to have high testability. Conversely, if corrupted inputs are subsumed or “cleansed” by the component such that correct outputs result, then the component will be observed to have low testability. Testability is a key metric for determining where testing resources should be focused [8].

Another important metric provided by IPA is the ability to measure the effectiveness of component “wrappers”. Component wrappers are used to thwart errors originating in one component from propagating into the rest of a system. Wrappers exercise algorithms to detect anomalous outputs and replace them with “correct” or “safe” outputs. The Generic Wrapper Toolkit project underway at Trusted Information Systems and funded by the Defense Advanced Research Projects Agency (DARPA) is an example of software-wrapping technology designed to improve security and reliability in distributed object systems [1]. Generic software wrappers use a Wrapper Definition Language (WDL) to define the security and reliability policies of components through their interfaces. The software wrappers inter-

cept interactions between components and bind functions specified in the WDL to enforce the security and reliability policies. Interface propagation analysis can be used to assess the effectiveness of component wrappers by numerically rating the propagation likelihood of corrupted outputs from a “wrapped” component. This analysis can be used to improve or “fine-tune” the functions coded in the WDL to provide higher levels of security and reliability.

Computing the interface propagation analysis metric is the first step in assessing the survivability of a distributed object system. This metric provides a measure for determining the likelihood of errors propagating throughout the system. If the system has low testability or is very tolerant to faults, components will likely suppress fault propagation. On the other hand, a system with high fault propagation may be very intolerant to component faults and may very well fail in the presence of a single component failure. In the strictest conservative sense, IPA provides a metric for computing system survivability. That is, if any error propagating outside of a component constitutes system failure, then IPA defines a method for assessing system survivability. On the other hand, if system survivability can be codified in a set of policies, statements, or assertions, then system survivability can be dynamically assessed in an application-specific manner.

## 4.2 System-level survivability analysis

Survivability relates a prediction that malicious or non-malicious anomalous events resulting from component behavior or external sources will not result in a loss of mission for a given system. As mentioned in the preceding section, if an incorrect output constitutes a loss-of-mission, then IPA can provide the survivability prediction. However, if the survivability of a given system can be expressed as a set of properties relating to the software or the system, then the survivability can be dynamically assessed through analysis. The survivability properties that are expressed as a set of assertions or predicates can be assessed in two distinct environments—component-level and system-level. At the component-level, survivability analysis will be able to capture inputs and outputs from a component under analysis. For example, if the difference between outputs of two subsequent cycles of a sampling control system is greater than a safe threshold amount, the predicate might be coded as:

$$PRED = |O(n) - O(n - 1)| > safe\_threshold.$$

This predicate computes a time series function on successive outputs from a component to determine if a safe threshold has been exceeded.

The second environment in which survivability properties can be assessed is at the system-level. The system-level survivability analysis uses information collected from component-level analysis to determine if the system-wide survivability properties have been violated. For example, if a survivable system is computing system functions in a redundant voting architecture, a system-level survivability condition would require a majority of the system objects to be operating. Using this information, system-wide assertions about survivability readiness can be evaluated to determine if the system as a whole is still in a survivable posture.

### 4.3 Covert Channels

It is important to recognize the limitations of component wrappers and IPA in detecting and restricting covert channels. Component wrappers can, in theory, restrict the propagation of unsafe outputs from a software component for *well-defined* outputs. However, wrappers have little or no ability to prevent the propagation of information from a component through a *covert* channel. That is, a component with a well-defined interface that is “wrapped” may still be able to send commands or data through covert channels that are not defined in the component’s interface. Since the source code for a COTS component probably will not be available for inspection or analysis, the wrapper can only attempt to control the interfaces which are defined. The ability to exploit covert channels can subvert component wrappers and permit malicious component behavior.

Since the IPA algorithm assesses the strength of the component wrapper (or interface) and has no extra knowledge of the component besides its well-defined interface, IPA cannot be used in its current form to determine the extent to which a component uses covert channels. The problem of covert interfaces is particularly acute in security-critical applications where malicious commands that are not visible to the external interface may be embedded within COTS components. To address this problem, we have developed a predictive analysis technique distinct from IPA for assessing the existence of covert channels which is not discussed in this paper.

## 5 Conclusions

In the mid 1980s, Japan’s Prime Minister, Yasuhiro Nakasone, predicted the Information Revolution that we are in the midst of today. He spoke of a future society, evolved out of past societies that had relied on barter, precious metals, and paper currency for trade, in which *information* would serve as currency. In this brave new world, Mr. Nakasone predicted that the ability to manipulate and access information would be the measure of wealth and power. Interestingly, the Internet, which provides the backbone of today’s Information Revolution, has become an important medium for financial transactions as well as the focus of many other forms of information manipulation and propagation. As such, the Internet and, more generally, the movement of commerce towards an electronic infrastructure, embodies the prediction of the Minister.

With information quickly becoming a measure of a nation’s wealth, the desire to control and conquer information ensues. This has led to the coining of terms such as “information warfare” and “information survivability.” Similar to traditional warfare, information warfare (IW) has two components: an offensive side, and a defensive side. Offensive IW stems to take away an adversaries access to information systems, whereas defensive IW (or what is also called “survivability”) stems to provide continual information systems support regardless of the problems that may arise. Our interests touch on both sides of the information warfare issue, however our experience to date has been mainly on the defensive IW side—in information survivability.

This paper has focused on assessing the survivability of a distributed object system in the presence of component failures. Our position is that behavioral assessment techniques such as fault injection are the best existing approaches for assessing the survivability of

complex systems—especially when such systems are not formally specified and are generally untestable.

## Acknowledgment

The authors are currently being funded by DARPA/ITO in Information Survivability under contract F30602-95-C-0282. THE VIEWS AND CONCLUSIONS CONTAINED IN THIS DOCUMENT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL POLICIES, EITHER EXPRESSED OR IMPLIED, OF THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY OR THE U.S. GOVERNMENT.

## References

- [1] L. Badger. Generic software wrappers for security and reliability. In *Proceedings of the 19th National Information Systems Security Conference*, pages 667–668, October 22-25 1996.
- [2] A.C. Carlson, W.R. Brook, and C.L.F. Haynes. Experiences with distributed objects. *AT&T Technical Journal*, pages 58–67, March/April 1996.
- [3] B.W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems*. Electrical and Computer Engineering. Addison-Wesley Publishing Company, Reading, MA, 1989.
- [4] N. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
- [5] N.D. Singpurwalla. The failure rate of software: Does it exist? *IEEE Transactions on Reliability*, 44(3):463–469, September 1995.
- [6] J. Voas, G. McGraw, A. Ghosh, and K. Miller. Gluing together software components: How good is your glue? In *1996 Proceedings of the Pacific Northwest Software Quality Conference*, to be published 1996.
- [7] J.M. Voas. Pie: A dynamic failure-based technique. *IEEE Trans. on Software Eng.*, 18(1):717–727, August 1992.
- [8] J.M. Voas and K. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.