

Achieving Critical System Survivability through Software Architectures

John C. Knight and Elisabeth A. Strunk

Department of Computer Science
University of Virginia
151, Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740, USA
(strunk|knight)@cs.virginia.edu

Abstract. Software-intensive systems often exhibit dimensions in size and complexity that exceed the scope of comprehension of system designers and analysts. With this complexity comes the potential for undetected errors in the system. While software often causes or exacerbates this problem, its form can be exploited to ameliorate the difficulty in what is referred to as a *survivability architecture*. In a system with a survivability architecture, under adverse conditions such as system damage or software failures, some desirable function will be eliminated but critical services will be retained. Making a system survivable rather than highly reliable or highly available has many advantages, including overall system simplification and reduced demands on assurance technology. In this paper, we explore the motivation for survivability, how it might be used, what the concept means in a precise and testable sense, and how it is being implemented in two very different application areas.

1 Introduction

Sophisticated hardware systems have been providing dependable service in many important application domains for some time. Systems such as electro-mechanical railway signals and aircraft hydro-mechanical controls are safety critical, and many strategies for achieving and analyzing dependability properties in these systems have been developed. Introducing software-intensive components into engineered systems, however, adds extra dimensions in size and complexity. Combining powerful computation facilities with high-speed digital communications can lead to systems that are thousands of times more complex than would normally be considered in a hardware system, such as the electronic funds transfer system within the financial network or the supervisory and control mechanisms within the electric power grid. Such systems are referred to as *critical infrastructure systems* because of the very high dependence that society now has on them.

Software also enables function to be implemented that would be impractical to implement in hardware. In *safety-critical embedded systems*, important facilities—including many that enhance safety—depend on complex software systems for correct operation. An example of this is stability augmentation in aircraft flight-control, where the digital system calculates a stream of small adjustments that must be made to control surfaces in addition to the pilot's commands.

Both critical infrastructure systems and safety-critical embedded systems can quickly exceed not only the scope of current approaches to analysis but also the comprehension capability of even talented, experienced system designers and analysts. When this happens, the potential for introducing undetected errors into the system is greatly increased. While software function causes or exacerbates some of the difficulties in system design, its form can be exploited to ameliorate them in what is referred to as a *survivability architecture*. In a system with a survivability architecture (referred to as a *survivable system*), the full set of system functions, though highly desirable, will not always be provided and need not be provided in order to prevent a catastrophic failure. Under certain adverse conditions that preclude the provision of total functionality, the system can offer an alternative service. This alternative service provides critical system functions, sacrificing some of the preferred service to ensure a level of continued operation that is considered acceptable even though it is not optimal.

Requiring that a system be survivable rather than reliable provides two major advantages. First, the amount of hardware replication required to meet hardware dependability goals can be reduced by architecting the system to allow and account for some unmasked failures; this confers considerable cost savings, particularly in large systems. Second, assurance of correct software function can be limited to the function whose correctness is crucial. Certain software faults can be tolerated by transitioning to providing only the crucial functionality, and the more limited size of the crucial functionality gives designers and analysts a much better chance of being able to cope with the complexity of the system.

In this paper, we examine the characteristics and dependability requirements of critical infrastructure and embedded systems. With these requirements in mind, we present the detailed definition of survivability and show how the definition can be applied. We then give examples of survivable systems and discuss the implementation of survivability using survivability architectures for both types of system.

2 Types of System

2.1 Critical Information Systems

Powerful information systems have been introduced into critical infrastructure applications as the cost of computing hardware has dropped and the availability of sophisticated software has increased [18]. Massive computerization has enabled efficiencies through tightened coupling of production and distribution processes. Just-in-time delivery of automotive parts by rail, for example, has enabled dramatic inventory reductions. Some forms of damage to these systems have no external effect because of appropriate redundancy; mirrored disks, for example, mask the effect of disk failure. But in other cases, the effects of damage will be so extensive that it will be visible to the system's users in the form of a lack of service or reduction in the quality of service. Events that disrupt critical infrastructure applications are inevitable; in practice, the continued provision of some form of service is necessary when damage precludes the provision of full service.

For the developer of a critical information system, knowing what service is required in the event that full service cannot be provided is very important. The current

notion of dependability for critical information systems does not provide the necessary concepts of degraded service and the associated spectrum of factors that affect the choice of degraded service as an explicit requirement. Survivability is the term that has come into use for this composite form. To provide a basis for a discussion and to motivate the definition, we enumerate the characteristics of infrastructure applications that affect the notion of survivability. The most important characteristics are:

- *System Size.* Critical information systems are *very* large. They are geographically diverse, topologically complex, and include large numbers of heterogeneous computing, storage and network elements.
- *Damage and Repair Sequences.* Events that damage a system are not necessarily independent nor mutually exclusive. A sequence of events might occur over time in which each event causes more damage. In effect, a bad situation gets progressively worse meaning that a critical infrastructure application might experience damage while it is in an already damaged state, and that a sequence of partial repairs might be conducted. A user might experience progressively less service over time as damage increases and progressively more as repairs are conducted.
- *Time-Dependent Damage Effects.* The impact of damage tends to increase with time. The loss associated with brief (seconds or less) interruptions of electric power, for example, can be mitigated in many cases. A protracted loss (days) is much more serious, with impact tending to increase monotonically with time.
- *Heterogeneous Criticality.* The requirements for dependability in infrastructure systems are considerable but some functions are more important than others and the importance of some functions often varies with time.
- *Complex Operational Environments.* The operating environments of critical infrastructures are of unprecedented complexity. They carry risks of natural, accidental, and malicious disruptions from a wide variety of sources.

2.2 Safety-Critical Embedded Systems

As with critical infrastructure systems, immense amounts of software have been introduced into safety-critical embedded systems for similar reasons and with similar results. There has been a shift towards digital implementation of many functions that used to be electro- or hydro-mechanical (fly by wire, for example), and many new service and software-intense safety functions have been introduced (enhanced ground proximity warning, for example).

All forms of damage to safety-critical systems must be anticipated and considered in system design and analysis. The emphasis in these systems has always been to mask the effects of faults, yet that is becoming increasingly difficult as the complexity of the systems increases. In addition, the overall complexity of the software upon which the systems rely has long surpassed the point at which comprehensive analysis is possible. Exploiting the notion of survivability for these systems can reduce system complexity and limit the amount of software that is crucial to dependable operation.

Mandated dependability requirements are set by regulating agencies such as the U.S. Federal Aviation Administration (FAA), Food and Drug Administration (FDA), and Nuclear Regulatory Commission (NRC). The FAA, for example, categorizes aircraft functionality into three levels of criticality according to the potential severity of

its failure conditions [14]. The most extreme criticality level is “**Catastrophic**: Failure conditions which would prevent continued safe flight and landing” [14]. Catastrophic failure conditions must be “extremely improbable” [14] or “so unlikely that [the failure condition is] not anticipated to occur during the entire operational life of all airplanes of one type” [14]. “Extremely improbable” corresponds to a quantitative failure rate of 10^{-9} failures per hour of operation.

Using survivability in systems of this type enables the extent of the system that is required to meet this extreme level of dependability to be reduced significantly. As in the critical infrastructure case, we enumerate the characteristics of safety-critical embedded systems that affect the notion of survivability. These characteristics are:

- *System Timing and Resource Constraints*. Embedded systems are often severely limited in power, cost, space, and weight, and so they are tuned to take the best possible advantage of their underlying resources.
- *System Coupling*. System components frequently have a very strong dependence on one another, and so the state of various system components must be seen individually and also as a whole when determining appropriate system behavior.
- *Damage and Repair Sequences*. Failures can occur in sequence, and while sometimes the system size allows reinitialization, at other times system criticality precludes any service interruption.
- *Heterogeneous Criticality*. Criticality of embedded system services also varies with function and with time. A medical application, for instance, is less dangerous during surgical planning than during the surgery itself.
- *Complex Operational Environments*. While the operational environments of embedded systems are fairly localized, they can still be affected by a variety of factors. Avionics systems are affected by factors such as weather, altitude, and geographic location.

3 What Is Survivability?

3.1 Existing Definitions of Survivability

Like many terms used in technologies that have not yet matured, several notions of survivability have appeared, and a rigorous definition has been presented only recently. Survivability has roots in other disciplines; for instance, a definition used by the telecommunications industry is:

Survivability: A property of a system, subsystem, equipment, process, or procedure that provides a defined degree of assurance that the named entity will continue to function during and after a natural or man-made disturbance; e.g., nuclear burst. Note: For a given application, survivability must be qualified by specifying the range of conditions over which the entity will survive, the minimum acceptable level or [*sic*] post-disturbance functionality, and the maximum acceptable outage duration [46].

The sundry definitions of survivability (see also, for example, [12, 13]) vary considerably in their details, but they share certain essential characteristics. One of these is the concept of service that is essential to the system. Another is the idea of damage that

can occur; and a third, responding to damage by reducing or changing delivered function. Further, the definitions used outside of computing introduce the idea of probability of service provision as separate from the probabilities included in dependability.

While these definitions offer a firm starting point for a definition, they offer only an *informal* view of what survivability means. This view is analogous to the colloquial view of reliability—that the system rarely or never fails. The *formal* view of reliability, on the other hand, states that a system is reliable if it meets or exceeds a particular probabilistic goal. It is the formal definition that provides criteria for reliability that can be tested. Likewise, a formal definition for survivability that provides testable criteria is needed. If a system’s survivability characteristics cannot be tested, then system developers cannot tell whether they have met users’ requirements. The informal definitions above, for example, do not specify: which functions are essential; under what fault conditions the essential functions will be provided; or the timing requirements on transitioning to provide only essential functions.

Knight et al. give a definition based on specification: “A system is survivable if it complies with its survivability specification” [20]. They draw on the properties mentioned above and present a specification structure that tells developers what survivability means in an exact and testable way. It is this definition that we characterize and build on here.

3.2 A More Precise Intuitive Notion

What has been implied so far, but has not been made explicit, is the idea of *value* provided by a system to its users. In the critical infrastructure case, this value is essentially an aggregate over time; provision of electric service, for instance, is not critical at a single point in time or even for a short length of time, but the service is crucial over the long term. In contrast, in many embedded systems provision of service at a particular point in time is paramount; a nuclear shutdown system, for example, is of no use if it is operational continuously only until it is called upon.

The core of the survivability concept is that value is not a Boolean variable. A system is not constrained to provide service or not; it can provide service whose value to the user is less than that of the system’s standard service, but still enough to meet critical user requirements. For an aircraft, such service might be basic control surface actuation without enhanced fly-by-wire comfort and efficiency algorithms.

The general idea of survivability is that a system will “survive” (i.e., continue some operation), even in the event of damage. The operation it maintains may not be its complete functionality, or it might have different dependability properties, but it will be some useful functionality that provides value to the users of the system, possibly including the prevention of catastrophic results due to the system’s failure. Such a strategy is used extensively in industrial practice, but it does not rest on a rigorous mathematical foundation and so the properties of such a system are not guaranteed.

One might want to pursue survivability of a system for two reasons. First, while many services that critical systems provide are helpful or convenient, not all of them are necessary. Dependability guarantees on the noncrucial services can be extremely expensive to implement, particularly in the case of infrastructure systems where thousands of nodes would have to be replicated were the system engineered to provide all

services with high dependability. In this case, survivability increases value delivered to the user because it decreases the cost of the services that are rendered.

Second, it might be the case that a system is so complex that there is no way to determine with sufficient confidence that the system meets its dependability goals. Cost can influence a decision to use survivability in this sense as well—sufficient assurance, if possible, could be extremely costly—but even more difficult is the question of validation. Validation is an informal activity [42], and must ultimately be performed by humans. The larger and more complex the system, the more difficulty humans have in determining whether it will meet their informal notion of what it should accomplish. If the system is survivable, human oversight need only ensure to ultradependable levels the crucial function and the transition mechanism, a simpler task than ensuring ultradependability of the entire system.

We now describe the different facets of the survivability concept that lead into a more rigorous definition:

- *Acceptable services.* A simple strategy for specifying a survivable subsystem is to define what constitutes the system's desired functionality and what constitutes its crucial functionality. The crucial functionality can then be ultradependable and the desired functionality *fail-stop* in the sense of Schlichting and Schneider [37]. This strategy is oversimplistic, however, for three reasons. First, the user is likely to expect some minimum probability that the full function is provided. Operating exclusively in backup mode is almost certain to be unacceptable. Second, there can be more than two major classes of function. If the system must degrade its services, some services are likely to be more valuable than others even if they are not essential, and the subsystem should continue to provide those services if possible. Third, the functionality that is determined to be crucial by domain experts will usually depend upon operating circumstances. As an example, consider an automatic landing system. It could halt and simply alert pilots of its failure if it were not in use (i.e., in standby mode), but if it were controlling an aircraft it would have to ensure that pilots had time to gain control of the situation before halting. The *set of acceptable services* contains those services which provide the best possible value to the user under adverse circumstances, and which take account of the three factors above.
- *Service value.* If we are to base design decisions on delivered value, it is important to have a precise characterization of that value. The metrics by which delivered value can be measured vary among applications, and can become incredibly complex. Absolute value is not important, however, because the purpose of a value metric is to decide what will provide the most benefit under specific circumstances: a quality that can be specified with relative values among the set of acceptable services. The relative value provided by a service can change under varying operational conditions, but for a system with a small number of separate services and a small number of salient environmental characteristics, the specification of relative service values can be done simply in a tabular form without having to conduct a detailed utility analysis.
- *Service transitions.* The system will provide only one member of the set of acceptable services at a time. Under normal circumstances, that member will be the *pre-*

ferred service which includes all desired functionality. If the preferred service can no longer be maintained, the system must transition to a different acceptable service. Which service it transitions to will depend on which services can still be provided and the operational conditions at the time of the transition. The set of valid transitions defines the specifications to which the system can transition from a particular specification. When a reconfiguration occurs, the service with the highest relative service value in that set for the prevailing operational conditions will be chosen. Transitions also might be triggered by a change in operational conditions as well as some form of damage if the system is not delivering its preferred service.

- *Operating environment.* Since different services provide different relative values under different operating conditions, the characteristics of the operating environment that affect relative service values must be enumerated, so that the environmental state can be determined and used in calculating the most appropriate transition to take. As an example, note that time of day has a serious impact on usage patterns of infrastructure services and can affect the risks associated with critical embedded systems such as aircraft and automobiles.
- *Service probabilities.* Users will demand that they have more than strictly crucial operational capability the vast majority of the time. Simply listing a set of acceptable services, then, is insufficient because that would imply that implementing only the basic service fulfills the specification. Rather, a set of minimal probabilities on value of delivered service are required. These probabilities are requirements on a system's meeting the dependability requirements of the operational specification. In other words, if the system were operating under the specification S_1 , then the probability would be that S_1 's dependability requirements were met. This is not the same as strict composition of dependability and survivability probabilities, because if the survivability probability is not met, then the system will transition to another specification with a new set of dependability probabilities. It might be the case that specification probabilities should be grouped. For example, if a system can transition from S_1 to either S_2 or S_3 depending on operating conditions, and $(S_2 \text{ OR } S_3)$ provides some coherent level of service, then the desired probability would be on $(S_2 \text{ OR } S_3)$ rather than the specifications individually. Generally, the same probability might be assigned to both S_2 and S_3 so that that probability would then hold over $(S_2 \text{ OR } S_3)$. Specifying that the probability hold over the disjunction, however, leaves more slack in the implementation of the system as well as being more intuitive.

3.3 Defining Survivability

Now that we have presented an informal explanation of the meaning and characteristics of survivability, we summarize the definition [20]. The definition is phrased as a specification structure; a system specification that has each of these elements is defined to be a survivability specification, and a system built to that specification has a survivability architecture. Such a specification contains six elements:

- S:** *the set of functional specifications of the system.* This set includes the *preferred* specification defining full functionality. It also includes alternative specifications representing forms of service that are acceptable under certain adverse conditions (such as failure of one or more system components). Each member of **S** is a full specification, including dependability requirements such as availability and reliability for that specification.
- E:** *the set of characteristics of the operating environment* that are not direct inputs to the system, but affect which form of service (member of **S**) will provide the most value to the user. Each characteristic in **E** will have a range or set of possible values; these also must be listed.
- D:** *the states of E that the system might encounter.* This is essentially the set of all modes (i.e., collection of states) the environment can be in at any particular time. Each element of **D** is some predicate on the environment. **D** will not necessarily be equal to the set of combinations of all values of elements in **E** since some combinations might be contradictory. Including **D** as a specific member allows completeness checks across environmental modes.
- V:** *the matrix of relative values each specification provides.* Each value will be affected both by the functionality of the specification and the environmental conditions for which that specification is appropriate. Quantifying these values is impossible, but using relative values gives the ordering needed to select a service based on prevailing conditions.
- T:** *the valid transitions from one functional specification to another.* Each member of **T** includes the specification from which the transition originates (source specification), the specification in which the transition ends (target specification), and a member of **D** defining the environmental conditions under which that transition may occur (the transition guard). The guard enables a specifier to define which transitions are valid under certain circumstances, and the developer can then use **V** to decide which target specification is most valuable under those conditions.
- In information systems this is sufficient because while there will be some approximate time constraint on the transition's occurrence, it is unlikely to be a very tight time bound. In embedded systems, on the other hand, the time required to transition often will be an integral part of the transition specification, and some global state invariant might need to hold during the course of the transition. Thus **T** has two optional members: the transition time, the maximum allowable time for a particular transition during which the system may be noncompliant with all members of **S**; and the transition invariant, which may not be violated at any point in which the system is noncompliant with all members of **S**.
- P:** *the set of probabilities on combinations of specifications.* Each member of **P** will be a set of specifications containing one or more elements that is mapped to a probability. The specifications in each set provide approximately the same level of functionality, under different environmental conditions. The probability is the probability of a failure in the system when the system is in compliance with one of the specifications (or the single specification, if there is only one in the set for that

probability). Each probability is a minimum; for example, a failure of the primary specification might in reality be extremely improbable, so that the system never transitions to another specification. The probabilities are set by the systems owners to document the required service guarantees and they serve to provide a lower-bound guarantee of system operation.

4 System Examples

4.1 An Example of Survivability in Critical Information Systems

To illustrate the definition, we present an example based on a hypothetical financial payment system. We assume a hierarchical network topology in which there are a large number of nodes associated with *branch* banks (small retail institutions), a smaller number of *money-center* banks that are the primary operations centers for major retail banking companies, and a small set of nodes that represent the *Federal Reserve Bank*. Examples of identified hazards to the system include major hardware disruption in which communications or server machines become non-operational; coordinated security attacks in which multiple commercial bank regional processing centers are penetrated; and regional power failure in which either many branch banks are disabled or several money-center banks are disabled.

For this example, we assume several forms of tolerable service for the payment system (shown in Fig. 1): (S_0) *Preferred*, including electronic funds transfers, check processing, support for financial markets such as stocks and bonds, and international funds transfers; (S_1) *Industry/Government*, which limits service to transfer of large sums among major industrial and government clients only; (S_2) *Financial Markets*, which defines service for all the major financial markets but no other client organizations; (S_3) *Government Bonds*, which defines service for processing of sales and redemptions of government bonds only and only by major corporate clients; and (S_4) *Foreign Transfers*, in which transfers of foreign currency into or out of the country are the only available service.

To decide upon the relative value seen by users of the payment system associated with the services in this example (V in Fig. 1), we note: (1) that settlement by the clearing houses and the Federal Reserve Bank occurs during the late afternoon; (2) domestic markets are closed at night; and (3) stock, bond, and commodity markets must be accommodated when trading volumes are exceptionally and unexpectedly high. There is little value to providing processing service to domestic financial markets overnight, for example, and thus international transfers of funds have higher value. Similarly, extreme volume in domestic financial markets leads to a high demand on the financial payment system and this demand must be met if possible. Finally, during times of political crisis, sentiment turns away from most traditional financial instruments and government bonds become heavily sought after. Thus, during such times, the ability to maintain a market in government bonds is crucial. The relevant members of E , then, are time of day, trading volume, and political climate. D is the powerset of members of E , except that: (1) all states where the political climate is unstable are grouped together, since their values are the same; and (2) high trading volume cannot occur during late afternoon or at night. P for this specification is the set of individual speci-

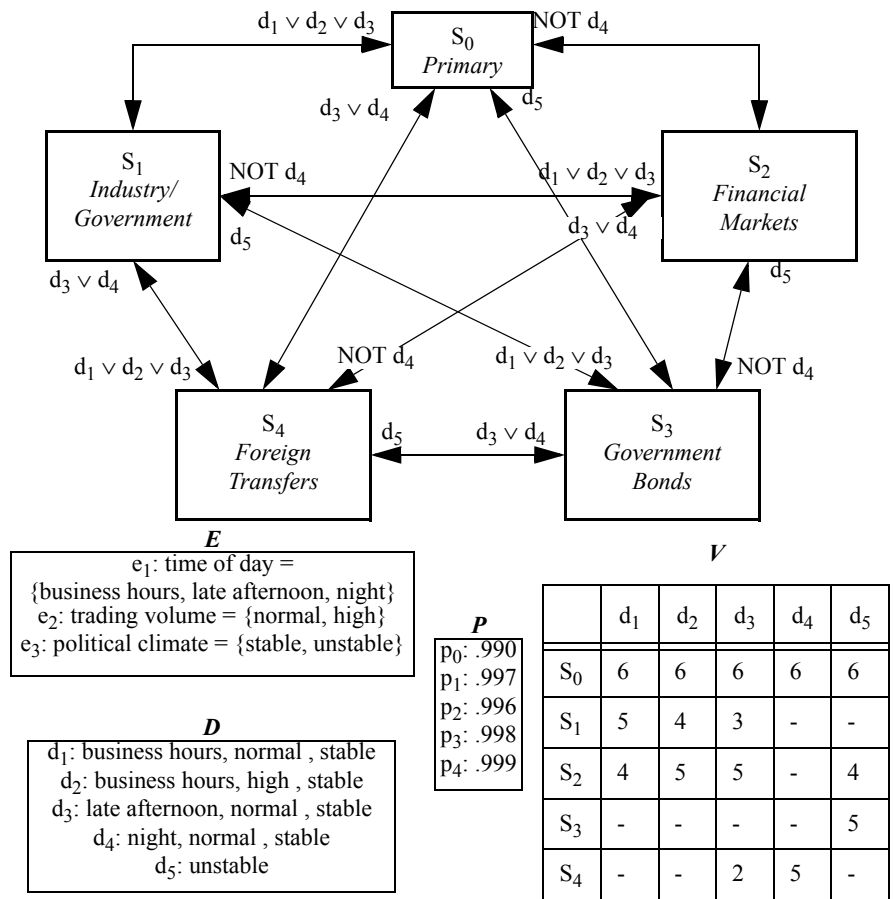


Fig. 1. Hypothetical Financial Payment System Survivability Specification

cation probabilities; composing specification probabilities does not make sense in this situation.

In our example, consider first the occurrence of a fault with a major file server that occurs during the middle of a normal market day (i.e., system state d_1) and which cannot be masked. To meet its survivability specification, the options that the system has are to transition to providing either service S_1 , S_4 , or S_2 , (see Fig. 1) and the maximum relative value to the user (from the V table indexed by the current conditions d_1) would be in service S_1 in this case. Were this to occur during the night, the transition would be to service S_4 because the current conditions would be d_4 . Now suppose that while the server is down, a coordinated security attack is launched against the system (a bad situation getting worse). In that case, the response to the attack might be to shut down as much of the system as possible. The system would transition to service S_3 since that would permit the best support in the event that the situation developed into a governmental crisis.

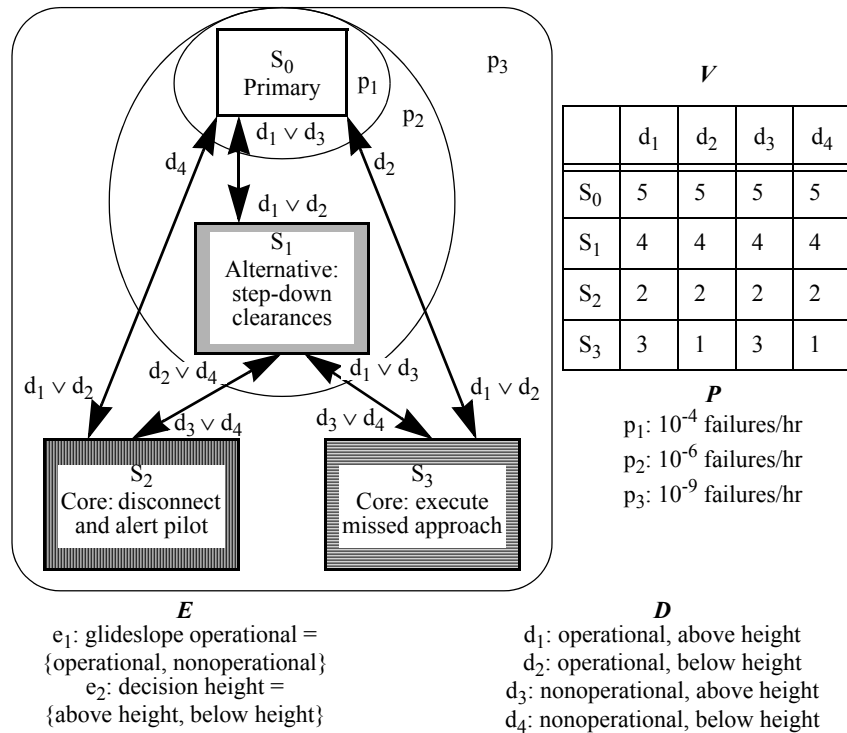


Fig. 2. Hypothetical ALS Survivability Specification

4.2 An Example of Survivability in Embedded Systems

As an illustration of how survivability might be applied to a safety-critical system, consider a hypothetical automatic landing system for a commercial aircraft. Assume four functional specifications, as shown in Fig. 2. The primary specification (S₀) will have all of the functionality the pilot desires for the system. The consequences of any failures will be minor because, if they have the potential to be more severe, the system can transition to one of the other three specifications. Therefore, using the FAA’s criticality levels (see section 2.2), any failure in the primary specification may be “probable” provided that failure of a transition to another specification is “extremely improbable”.

The first alternative specification (S₁) will have much of the functionality desired by the user, but some desirable yet unnecessary functionality removed. For example, the system might have to follow the step-down altitude clearances for the runway to descend at the proper rate rather than using the glideslope. All failures in this specification must be “improbable”; its functionality is important enough that frequent interruptions could have adverse consequences. However, provided that failure of a transition to another specification is “extremely improbable”, none of it need be

“extremely improbable” because any failures with potentially catastrophic consequences will cause a transition to a different alternative specification (S_2 or S_3).

S_2 and S_3 are the specifications that have very high dependability requirements. We will let S_2 be the specification requiring that the system disconnect and alert the pilot while remaining on course if the system fails and S_3 be the specification requiring the aircraft to execute a missed approach and alert the pilot on system failure. They contain the minimum functionality necessary to maintain safe operation of the system. Any non-masked failure of either of these specifications—such as failure to alert the pilot that the system has malfunctioned and the pilot is now in control—must be “extremely improbable”, as the specifications are designed to include only the system functionality whose failure could have catastrophic consequences.

Whether the system transitions to S_2 or S_3 on a failure of S_1 depends on whether the aircraft is above or below decision height at the time of the transition, based on the assumption that presenting the possibility of a go-around is more valuable under those circumstances. The new probability requirement, then, would be that a failure of S_2 *above* decision height is “extremely improbable”, and a failure of S_3 *below* decision height is “extremely improbable”. In some cases the environmental conditions might change, and a transition between specifications appropriate to different conditions must occur in order to keep the system operating with its optimal functionality.

5 Implementation of Survivable Systems

5.1 The Role of Fault Tolerance

Survivability is a system property that can be required in exactly the same way that the other facets of dependability can be required. There is no presumption about how survivability will be achieved in the notion of survivability itself—that is a system design and assessment issue. However, the probabilities associated with each of the tolerable forms of service are important design constraints since they will determine which design choices are adequate and which are not.

A practical survivability specification will have achievable probabilities and carefully selected functionality specifications. Thus, in such a system, the effects of damage will not be masked necessarily; and, provided the probabilities are met in practice, degraded or alternative service will occur. In effect, this implies that the survivability requirement will be achieved by the fault-tolerance mechanism, i.e., the system will have a fault-tolerant design. Note, however, that the N different functions in the survivability specification do not correspond to functions that can be achieved with the resources that remain after N different faults. The N functions in the survivability specification are defined by application engineers to meet application needs and bear no prescribed relationship to the effects of faults. Many different faults might result in the same degraded or alternative application service. The role of a survivability architecture is to provide the necessary system-level framework to implement the fault tolerance necessary to meet the system’s survivability goal. In the next two sections, we discuss two examples.

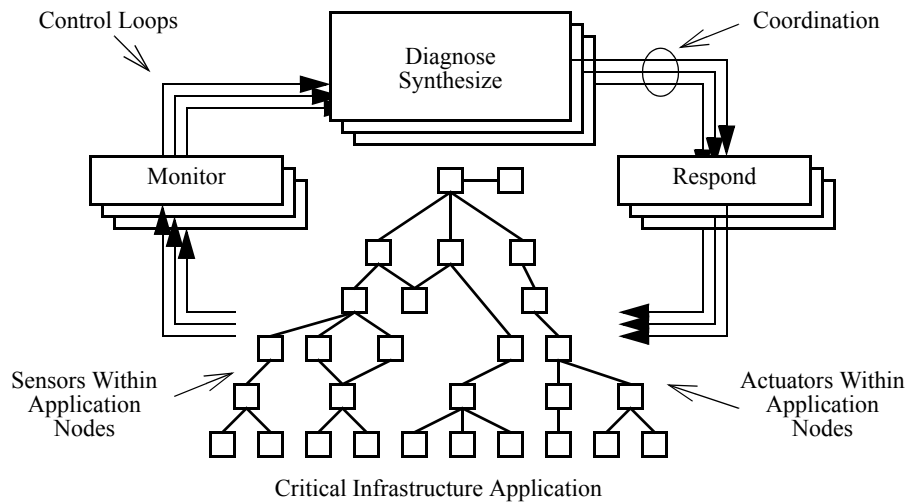


Fig. 3. The Willow Reactive System.

5.2 The Willow Reactive Mechanism

The fundamental structure of the Willow architecture [19] is a set of *control loops* each of which has monitoring, diagnosis, synthesis, coordination, and response components [45]. These components provide application-level fault tolerance. Monitoring and diagnosis provide error detection and synthesis; coordination and response provide error recovery. The overall structure is depicted in Fig. 3. The control loops begin with a shared *sensing* capability shown within the application nodes. Sensors can include reports from application software, application heartbeat monitors, intrusion detection alarms, or any other means of measuring actual application properties.

From sensing events, independent *diagnosis and synthesis* components build models of application state and determine required application state changes. Synthesis components issue their intended application changes as *workflow* requests. These are coordinated by the workflow and resource managers to ensure that changes occur correctly and smoothly within the application. When workflows are allowed to activate, workflow events are received by the application nodes and result in local system state changes. *Actuation* completes the control loop cycle.

As an example of the way in which the Willow reactive mechanism might be used, consider the hypothetical financial system introduced in section 4.1. The system might consist of several hundred servers of different types and tens of thousands of clients all communicating via a private network. Such a system provides services that are critical to several communities, and both security and availability are important properties. The Willow reactive mechanism could be used to deal with a variety of faults, both malicious and non-malicious, that might affect the system. A control loop could be deployed to deal with node failures that might occur because of hardware or software failures. A second control loop could be deployed to deal with coordinated security attacks. We discuss how these loops might work in the remainder of this section.

Dealing with complex faults

The reactive controller is a fully automatic structure that is organized as a set of finite state machines. The detection of the erroneous state associated with a fault (i.e., error detection) is carried out by a state machine because an erroneous state is just an application system state of interest. As the effects of a fault manifest themselves, the state changes. The changes become input to the state machine in the form of events, and the state machine signals an error if it enters a state designated as erroneous. The various states of interest are described using predicates on sets that define part of the overall state. The general form for the specification of an erroneous state, therefore, is a collection of sets and predicates. The sets contain the application objects of concern, and the predicates range over those sets to define states either for which action needs to be taken or which could lead to states for which action needs to be taken.

In the financial system example, events occurring at the level of individual servers or clients would be sent to the diagnosis and synthesis element of the Willow system where they would be input to a finite-state machine at what amounts to the lowest level of the system. This is adequate, for example, for a fault such as one causing commercial power to be lost. Dealing with such a fault might require no action if a single server or some small number of clients is affected because local equipment can probably cope with the situation. If a serious power failure affects a complete critical data center along with its backups, then the system might need to take some action. The action taken would be determined by the system's survivability specification as discussed in section 4.1. Recognizing the problem might proceed as follows. As node power failures are reported so a predefined set maintained by the diagnosis element, say *nodes_without_power*, is modified. When its cardinality passes a prescribed threshold, the recognizer moves to an error state. Once the recognizer is in the error state, the response mechanism would generate the appropriate workflows and individual nodes throughout the system would undertake whatever actions were defined by the workflows.

Sometimes, damage to a system is more complex. For example, a set of financial system nodes losing power in the West is the result of one fault, a set losing power in the East is the result of a second, but both occurring in close temporal proximity might have to be defined as a separate, third fault of much more significance. Such a fault might indicate a coordinated terrorist attack or some form of common-mode hardware or software failure. No matter what the cause, such a situation almost certainly requires a far more extensive response. We refer to such a circumstance as a *fault hierarchy*. A fault hierarchy is dealt with by a corresponding hierarchy of finite-state machines. Compound events can be passed up (and down) this hierarchy, so that a collection of local events can be recognized at the regional level as a regional event, regional events can be passed up further to recognize national events, and so on.

A coordinated security attack launched against the example financial system might include a combination of intrusions through various access points by several adversaries working at different locations, targeted denial-of-service attacks, and exploitation of previously unknown software vulnerabilities. Detection of such a situation requires that individual nodes recognize the circumstances of an attack, groups of nodes collect events from multiple low-level nodes to recognize a wide-area problem,

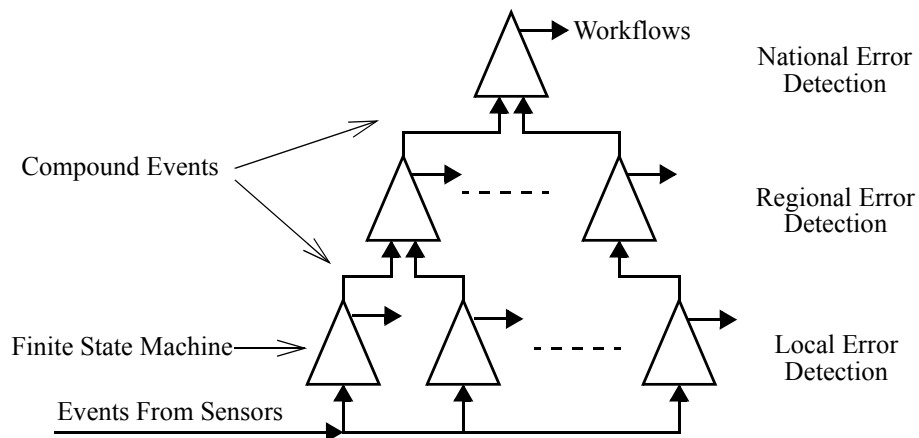


Fig. 4. Recognizing a Fault Hierarchy

and the high-level error-detection mechanism recognize the variety of simultaneous wide-area problems as a coordinated attack.

Provided sensor data was available from elements such as host intrusion-detection systems, network traffic monitors, and liveness monitors, a scenario such as this could generate an appropriate set of events as the different elements of the attack took place. A finite-state-machine hierarchy that might be used in such circumstances is shown in Fig. 4. As sensor events arrive from intrusion detection systems, an initial indication of the severity of the situation would be created. Local error detection might show that multiple clients were reporting intrusions. Each one might not be serious in its own right, but the temporal proximity of several would be a concern. The finite state machines receiving these intrusion events might not act on them other than to shut down the nodes reporting intrusions. Instead, they would generate compound events that would be sent to finite machines detecting regional security problems.

Regional error detection might determine that action needed to be taken after several local error detectors signalled a problem by forwarding a compound event. The action taken at the regional level might be to immediately switch all nodes in the region to a higher security level and limit service. If network traffic monitors then detected traffic patterns that were indicative of a denial of service attack, the error detection mechanism could trigger a national response.

Communication

The communication challenges presented by the Willow reactive system are considerable because of the scale of the networked applications that it seeks to address. The greatest challenge comes from the need to send reconfiguration commands to sets of nodes (those that have to act in the face of damage or an attack) where the relevant set is determined by analysis of the state.

The obvious way to approach such a requirement is for the Willow reactive system to maintain a central database with details of all the nodes in the system—how they are configured, what software they are running, their network addresses, and so on. This is completely infeasible for systems of the type we address. With hundreds of thousands of nodes, the size of the database would be prohibitive and it would constantly be in danger of being out of date—network changes would not be reflected in the database until some time after they occurred.

To send reconfiguration commands, the Willow system uses a novel communications approach called *selective notification*, an event communication mechanism combining content-based addressing, intentional addressing, and sender qualification in a unified structure for the delivery of events [33]. It has three primary components: (1) *symmetric decoupled communication* that combines content, sender, and receiver addressing in a single property-based addressing language; (2) *descriptive communication policies* in which communication relationships are defined at the level of policies constraining properties of objects relevant to communication; and (3) *simultaneous addressing* in which content-based, intentional, and sender-qualified addresses are applied simultaneously in determining the delivery of each event.

Returning to the example security attack on the hypothetical financial system, the actions needed for defense as the attack develops would be communicated to the nodes that need to take action by selective notification. The regional response of shutting down all nodes signalling intrusions and switching all other nodes in the affected region to a higher security level would be effected by two uses of selective notification. In the first, a command to shut down would be transmitted to “All Nodes In Region X With Triggered Intrusion Detection Alarms” where “X” is identity of the affected region. In the second, a command to change security parameters would be sent to “All Operating Nodes In Region X”. The phrases in quotations are the addresses used by selective notification. By selecting nodes based on properties, only essential network state information needs to be maintained by the Willow system.

The Willow reactive system has been evaluated as a means of defending against fast-moving worms. Using a single control loop implementation, worm propagation has been studied by simulation and the effect of a Willow-based defense system assessed. The details of that study can be found in the work of Scandariato and Knight [36].

Dealing with conflicting goals

The Willow reactive system consists of multiple asynchronous control loops, and each could initiate reconfiguration at any time. Clearly, this means that either all but one has to be suspended or there has to be a determination that they do not interfere. In the financial system example, we hypothesized that there might be two control loops and it is clear that they could easily conflict. If one were reconfiguring the information system to deal with a common-mode software failure when a security attack was initiated, it would be essential to take whatever actions were deemed necessary to counter the effects of the security attack. Reconfigurations underway might have to be suspended or even reversed. In such a situation, unless some sort of comprehensive control is exercised, the system can quickly degenerate into an inconsistent state.

One approach would be to have each source make its own determination of what it should do. The complexity of this approach makes it infeasible. An implementation would have to cope with on-the-fly determination of state and, since initiation is asynchronous, that determination would require resource locking and synchronization across the network.

The approach taken in Willow is to route all requests for reconfiguration through a resource manager/priority enforcer. The prototype implementation uses predefined prioritization of reconfiguration requests and dynamic resource management to determine an appropriate execution order for reconfiguration requests. It does this using a distributed workflow model that represents formally the intentions of a reconfiguration request, the temporal ordering required in its operation, and its resource usage. Combined with a specified resource model, this information is the input to a distributed scheduling algorithm that produces and then executes a partial order for all reconfiguration tasks in the network.

Scheduling is preemptive, allowing incoming tasks to usurp resources from others if necessary so that more important activities can override less important ones. Transactional semantics allow preempted or failed activities to support rollback or failure, depending on the capabilities of the actuators that effect the reconfiguration.

5.3 Embedded System Reconfiguration

Turning now to embedded systems, the notion of reconfiguration to deal with faults has been used extensively in safety-critical and mission-critical systems. For example, the Boeing 777 uses a strategy similar to that advocated by Sha's Simplex architecture in which the primary flight computer contains two sets of control laws: the primary control laws of the 777 and the extensively tested control laws of the 747 as a backup [38]. The Airbus A330 and A340 employ a similar strategy [41] as have embedded systems in other transportation, medical, and similar domains. Existing approaches to reconfigurable architectures are, however, ad hoc; although the system goals are achieved, the result is inflexible and not reusable. Another important issue is the difficulty of achieving the necessary level of assurance in a system that has the capability of reconfiguring.

A prototype experimental survivability architecture for embedded systems designed to deal with these issues is illustrated in Fig. 5. In this architecture, these subsystems interface with the Subsystem Control Reconfiguration Analysis and Management (SCRAM) middleware. The SCRAM layer interfaces with the host operating system and various error-detection mechanisms deployed throughout the system.

The goal of this architecture is to permit the entire system to be survivable and consequently to provide the service required for safety (but not necessarily any other services) with a very high level of assurance. System survivability is achieved: (a) by ensuring that subsystems possess certain crucial properties; (b) by precise composition of the properties of the individual subsystems; (c) by controlled reconfiguration of subsystems if they experience local damage; and (d) by controlled reconfiguration at the system level if necessary.

Each subsystem is constructed individually to be a survivable entity and to provide a set of acceptable services in the sense of the survivability framework described

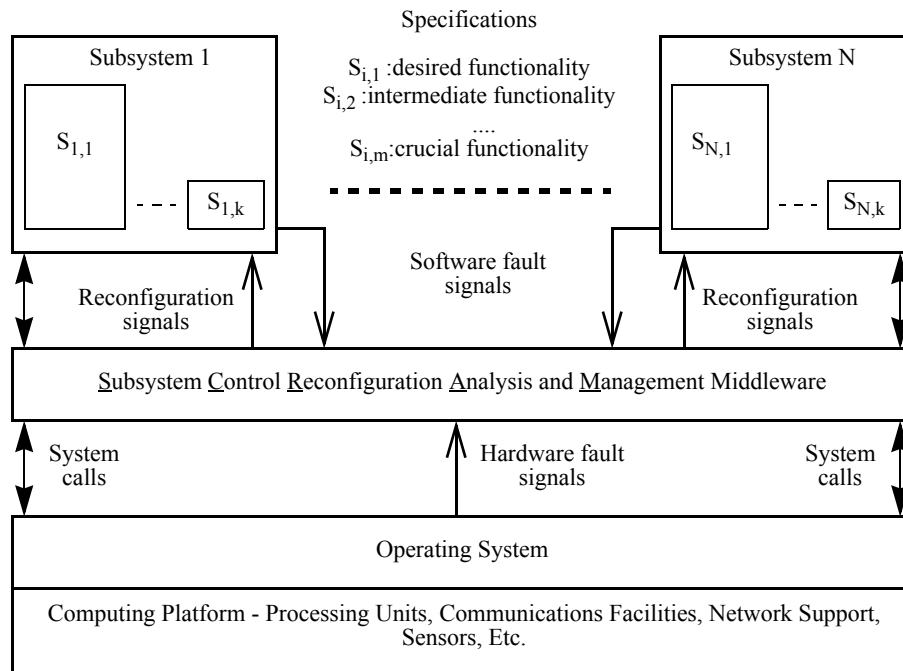


Fig. 5. Survivable Embedded System Architecture Overview

above. During operation, the SCRAM layer is responsible for reconfiguring the system so as to ensure the continued provision of essential aspects of the system's functionality. Reconfiguration is initiated by any sequence of events that either preclude the system from providing full functionality or indicate that providing full functionality would be unwise. An example of the former would be the defective operation or loss of any major system component, and an example of the latter would be an anticipated loss of power or communications in the near future. These events are detected by various means, including checking mechanisms within the different software subsystems themselves.

Protection Shells and Fail-Stop Software

The argument for the efficacy of survivability in embedded systems is based on the notion that in practice, many functions in safety-critical systems do not need to be ultrareliable, they need to be fail-stop [37]. In other words, it is sufficient for the function to either work correctly or to stop and signal that it has failed. As an example, consider the hypothetical automatic landing system (ALS) from Section 4.2. Although part of the overall avionics system, the ALS could be defective during cruise without posing a serious threat to the aircraft. Provided the ALS either works correctly or stops and alerts the pilot, the aircraft is unlikely to come to harm.

The software analog of fail-stop machines is the concept of *safe programming* introduced by Anderson and Witty [2]. The concept of safe programming is (in part) to

modify the postcondition for a program by adding an additional clause stating that stopping without modifying the state and signalling failure is an acceptable result of execution. A *safe* program in this sense is one that satisfies its (modified) postcondition. The problem of assurance has thus been reduced to one of assuring comprehensive checking of the program's actions rather than assuring proper overall functionality.

Related to safe programming, and providing a mechanism for implementing it, is the idea of a *safety kernel*. The idea has been studied in various contexts. Knight has introduced the term *protection shell* to more accurately describe what a small, simple policy enforcement mechanism for safety should be. The term "shell" implies that, rather than basing safety policies on what information can pass to and from the processor, the policies should be based on what outputs can be returned to the hardware controlled by the application.

Protection shells can be used to implement fail-stop function in survivable subsystems. Less critical components can have shells that guarantee the entire piece of function is fail-stop, while crucial components' shells can guarantee fail-operational capability or provide assurance that failures are acceptably improbable. Whichever capability is needed, the analysis associated with the shell will be much simpler than that associated with the full system because the shell for each component is much simpler than the component itself and is explicitly designed to facilitate that analysis.

The software subsystems that interact with the SCRAM layer can leverage the simpler analysis that protection shells afford to achieve the goal of obtaining a high level of assurance that the subsystem will, in fact, be survivable. Each major software component in a subsystem will be encapsulated in a shell that is geared to provide some set of guarantees for each possible operational level (individual specification). They can then be combined into a survivable subsystem structure using a standard architecture that provides subsystem reconfiguration and an interface allowing the SCRAM middleware to control subsystem internal reconfiguration.

Assurance Characteristics of a Reconfigurable System

The mechanism through which embedded system reconfiguration is achieved is complex, and going about its development in an ad hoc way could lead to lower overall system dependability than would have been exhibited by a system that could not reconfigure. Furthermore, dependability is a characteristic that typically must be met with very high assurance, and it cannot be assured without a rigorous characterization of this assurance.

We have described single-process reconfiguration informally as "*the process through which a system halts operation under its current source specification S_i and begins operation under a different target specification S_j* " [43]. From the definition of survivability, we know that a survivable embedded system has:

- A set S : $\{S_1, S_2, \dots, S_n\}$ of service specifications of the system
- A set E of possible environmental factors and their values
- The maximum time T_{ij} allowable for reconfiguration
- An invariant Inv_{ij} that must hold during reconfiguration

Using these concepts, reconfiguration can be defined as the process R for which [43]:

1. R begins at the same time the system is no longer operating under S_i
2. R ends at the same time the system becomes compliant with S_j
3. S_j is the proper choice for the target specification at some point during R
4. R takes less than or equal to T_{ij} time units
5. The transition invariant holds during R
6. The precondition for S_j is true at the time R ends
7. The lifetime of R is bounded by any two occurrences of the same specification

This is still an informal characterization of reconfiguration; a more thorough characterization upon which a rigorous argument can be built is available elsewhere [43].

Using the Software Architecture to Facilitate Proof

For large, complex systems, showing that an implementation has the characteristics outlined above can be a daunting task. Building in the potential for arguments of reconfiguration assurance in a more detailed software architecture can facilitate the creation of assurance arguments, just as the survivability architecture is designed to facilitate arguments of overall system dependability.

Software architectures have an advantage over software design in that they are general enough to be reusable across a number of systems. We argue that if we can show overall assurance properties of an architecture, we no longer have to show those assurance properties for each individual system that employs the architecture. Thus, assurance of such a system is defined as assurance of compliance with the architecture rather than assurance of overall reconfiguration.

Protection shells are a useful basic component of such an architecture in that they can export an interface whose properties can be assured by comprehensive checking of outputs if necessary. In addition to providing such functional properties, shells can be used to ensure invariants are held by restricting access to the interface. They can include definitions of preconditions and postconditions of module data, and call module functions that ensure preconditions and postconditions are met at appropriate times. Finally, each interface function of a shell can carry strict timing guarantees of its operation.

In our architecture, each function in a module interface presents a set of functional service levels. Reconfiguration to a similar but degraded service is accomplished by calling degraded versions of the same functions. The modules (and their shells) are linked through a *monitoring layer*, the architectural component that knows which member of S must operate, and passes this information along to the modules hierarchically to effect reconfiguration of function. Any detected and unmasked fault during computation of a module function causes control to be returned to the monitoring layer. The layer then activates the *reconfiguration mechanism*, which will choose the target specification, cause the data elements of the individual modules to conform to the precondition of the target specification, and instruct the monitoring layer to restart operation under the new specification.

Our architecture is designed to facilitate reconfiguration of a single application. A more extensive characterization of its structure and applicability is available

elsewhere [43]. We have outlined above how multiple applications can be combined using the SCRAM layer into an overall reconfigurable system. A similar assurance structure for the set of interacting systems is currently in development.

6 Related Work

Many concepts in the field of dependability are similar or related to the notion of survivability. In addition, many techniques for improving dependability are related to survivability architectures. We review this related work in this section.

6.1 Related Concepts

- *Dependability*

Avizienis et al. argue that survivability and dependability are equivalent—“names for an essential property” [4] although that statement was based on an earlier, informal definition of survivability [13]. Given the extremely broad definition of dependability, it is possible to argue that survivability is subsumed by dependability. However, survivability meets a need not adequately addressed by the standard definition of dependability since the latter’s breadth includes no structure and suggests a single-service view. Survivability emphasizes the need to specify systems that can provide different forms of service, each with its own complete set of dependability requirements, under different conditions. The problem with the single-service view is that it might not be cost effective to provide assurance of full service across the entire spectrum of potential damage; in a survivable system, a narrower subset of damage is selected to be addressed by the dependability requirements. Whether survivability is seen as a facet of dependability, or a composite of functional and dependability requirements, is a matter of definition. We use it as a composite of dependability requirements, i.e., a set of specifications each of which has its own dependability requirements in the traditional sense. This allows the standard technical notion of dependability to remain unchanged while adding another means of addressing informal dependability requirements on system function.

- *Graceful degradation*

The general notion of graceful degradation is clearly related to survivability. One definition, from the telecommunications industry, is: “Degradation of a system in such a manner that it continues to operate, but provides a reduced level of service rather than failing completely” [46]. According to this definition, survivability is a specific form of graceful degradation where reduced levels of service are specified and analyzed rigorously. Other perspectives on graceful degradation (e.g., Shelton et al. [40] and Nace and Koopman [28]) choose not to assume a specific set of alternative services, but rather determine the best possible value provided on-the-fly based on stated utility values of available functional components. Graceful degradation and survivability address a similar problems, but in subtly different ways. Graceful degradation attempts to provide the maximum value possible given a certain set of working functional components, which means it provides fine-grained control of functionality each form of which is determined dynami-

cally. Survivability, on the other hand, supplies only prescribed functionality sacrifices some of graceful degradation's postulated utility in order to provide tight control over the analysis and certification process of software. A gracefully degrading system is likely to degrade in steps as resources are lost, but those steps are not necessarily determined at design time. A survivable system degrades in explicit, tightly-controlled steps that provide predictability and the opportunity for stronger formal analysis at the expense of some potential utility.

- *Quality of service*

The telecommunications industry also has a definition for quality of service: "Performance specification of a communications channel or system" [46]. Quality of service (QoS) focuses on providing the most value with available resources and, like survivability, does this by incorporating some number of discrete functional levels. However, the term is generally used to refer to specific aspects of a system, for instance video quality or response time. QoS could be used by a survivable system, but survivability typically has a much broader impact on system function, changing the function more dramatically or replacing it altogether.

- *Performability*

The concept of performability is related in a limited way to survivability [26]. A performability measure quantifies how well a system maintains parameters such as throughput and response time in the presence of faults over a specified period of time [27]. Thus performability is concerned with analytic models of throughput, response time, latency, etc. that incorporate both normal operation and operation in the presence of faults but does not include the possibility of alternative services. Survivability is concerned primarily with system functionality, and precise statements of what that functionality should be in the presence of faults.

6.2 Related Embedded Architectures

Some architectural aspects of survivable embedded systems have been discussed by other researchers. In this section, we review two particular approaches: the simplex architecture and safety kernels.

- *Simplex architecture*

The Simplex architecture of Sha et al. [39] uses a simple backup system to compensate for uncertainties of a more complex primary system. The architecture assumes analytic redundancy: that two major functional capabilities with some significant design difference between them are used. The examples given primarily involve control systems. A survivability architecture is similar, but: (1) takes a more general view of possible user requirements, allowing more than two distinct functions; (2) uses tighter component control, disallowing software replacement (as opposed to reconfiguration) online in order to facilitate stronger analysis; and (3) addresses the problems associated with non-independence of software failures without requiring knowledge of completely separate methods to accomplish a goal (which might be an unreasonable assumption in many digital systems).

- *Safety Kernels*

The idea of a safety kernel derives from the related concept of a security kernel. Both are related to survivability since they have a goal of assuring that certain important properties (safety or security) hold even if functionality has to be abandoned. Leveson et al. use the term *safety kernel* to describe a system structure where mechanisms aimed to achieve safety are gathered together into a centralized location [22]. A set of fault detection and recovery policies specified for the system is then enforced by the kernel. Subsequently, Rushby defined the role of a safety kernel more precisely based on the maintenance of crucial properties when certain conditions hold [34]. In a discussion about the implementation issues of safety kernels, Wika and Knight introduced the idea of *weakened properties*, properties that are not checked in the kernel, but which the kernel ensures are checked by the application [53]. Weakened properties compromise between the need for assurance and the need for kernel simplicity. Burns and Wellings define a safety kernel as a *safe nucleus* and a collection of *safety services* [8]. The safe nucleus manages safety properties computing resources; the safety services check safety and timing invariants of individual applications. The safety services evade the problem of enforceability of only negative properties; including them means that all computation requests of an application can be monitored to check that safety assertions hold. Peters and Parnas [29] use the idea of a *monitor* that checks the physical behavior of a system by comparing it with a specification of valid behaviors. They explore the imprecision in a system's ability to detect its precise physical state and how this relates to the properties that must be checked by the monitor. These ideas apply to the safety kernel concept as well, as there may be some slack in safety policies or known imprecision in the physical system.

6.3 Fault-Tolerant Distributed Systems

A very wide range of faults can occur in distributed systems and fault tolerance in such systems has been an active area of research for many years. In this section, we mention briefly some of this research. For a more comprehensive overview, see Gartner [16] and Jalote [17].

Cristian surveyed the issues involved in providing fault-tolerant distributed systems [10]. He presented two requirements for a fault-tolerant system: (1) mask failures when possible; and (2) ensure clearly specified failure semantics when masking is not possible. The majority of his work, however, dealt with the masking of failures. Birman introduced the "process-group-based computing model" [7] and three different systems—ISIS, Horus [48] and Ensemble [49]—that built on the concept. In a recent system design, Astrolabe [50], Birman introduced a highly distributed hierarchical database system that also has capabilities supporting intentionally addressed messaging. By using a gossip-based communications protocol, Astrolabe organizes a hierarchical database of aggregated information about large-scale distributed computer systems. A virtual collection of high-level information contains highly aggregated information about the system as a whole, while lower level 'nodes' contain more localized information about distributed sub-systems.

In the WAFT project, Marzullo and Alvisi are concerned with the construction of fault-tolerant applications in wide-area networks [1]. The Eternal system, developed by Melliar-Smith and Moser, is middleware that operates in a CORBA environment, below a CORBA ORB but on top of their Totem group communication system. The primary goal is to provide transparent fault tolerance to users [25]. Babaoglu and Schiper are addressing problems with scaling of conventional group technology. Their approach for providing fault tolerance in large-scale distributed systems consists of distinguishing between different roles or levels for group membership and providing different service guarantees to each level [5]. Finally, the CONIC system developed by Kramer and Magee addresses dynamic configuration for distributed systems, incrementally integrating and upgrading components for system evolution. The successor to CONIC, Darwin, is a configuration language that separates program structure from algorithmic behavior [23, 24].

Kaiser introduced KX, Kinesthetics eXtreme, an architecture supporting the distribution and coordination of mobile agents to perform reconfiguration of legacy software systems [47]. The architecture allows mobile code (or SOAP message enabled code) to travel around a network and coordinate activities with one another through the Workflakes distributed workflow engine, which is in turn built atop the COUGAAR distributed blackboard system.

6.4 Intrusion-Tolerant Systems

The notion of Intrusion Tolerance emerged in the 1990's as an approach to security in which reliance would not be placed totally on preventing intrusion. Rather systems would be engineered to detect intrusions and limit their effect. Intrusion tolerance is closely related to fault tolerance for a specific type of deliberate fault, and provides a form of survivability.

Two major projects of note in the area are OASIS and MAFTIA. For detailed discussions of intrusion tolerance, see the text by Lala [21] and the report by Powell and Stroud [30]. For a discussion of the architectural issues, see the report by Verissimo et al [51]. We summarize some of the major research concepts here.

Some of the research in the field of intrusion tolerance has addressed important basic issues that can be viewed as building blocks for intrusion-tolerant systems. Examples include communications protocols that provide important quality guarantees [6, 31], approaches to the secure use of mobile code [3], file systems that resist or tolerate attacks [44], software wrappers that permit legacy code to have certain important quality attributes retrofitted [15], security protocols that help ensure certain security properties [11], mechanisms for dealing with buffer overflow attacks, and approaches to the creation of a secure certification authority [54].

As well as building blocks, experimental systems have been designed and built to demonstrate system-level concepts and techniques. Examples include HACQIT—a system that provides Internet services to known users through secure connections [32], ITDOS—an intrusion-tolerant middleware system based on CORBA [35], and SITAR—an intrusion tolerant system based on COTS servers [52].

7 Summary

The notion of survivability is emerging as an important concept in a number of areas of computer system design. In this paper, we have explored the motivation for survivability, how it might be used, what the concept means in a precise and testable sense, and how it is being implemented in two very different application areas.

Making a system survivable rather than highly reliable or highly available has many advantages including overall system simplification and reduced demands on assurance technology. Both of these advantages contribute to the potential for building systems and with assured operation that would otherwise be infeasible. Although service to the user will be limited during some periods of operation for a system that is survivable, the potential for overall cost-effective system development will often outweigh this limitation.

Acknowledgements

We thank Jonathan Rowanhill and Philip Varner for their significant contributions to the design and implementation of the Willow reactive system. This work was supported in part by NASA Langley Research Center under grants numbered NAG-1-2290 and NAG-1-02103. This work was supported in part by the Defense Advanced Research Projects Agency under grant N66001-00-8945 (SPAWAR) and the Air Force Research Laboratory under grant F30602-01-1-0503. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the Air Force, or the U.S. Government.

References

- [1] Alvisi, L. and K. Marzullo. "WAF: Support for Fault-Tolerance in Wide-Area Object Oriented Systems." *Proc. 2nd Information Survivability Workshop*, IEEE Computer Society Press, Los Alamitos, CA, October 1998.
- [2] Anderson, T., and R. W. Witty. "Safe programming." *BIT* 18:1-8, 1978.
- [3] Appel, A. "Foundational Proof-Carrying Code." IEEE Symposium on Logic in Computer Science, Boston MA, 2001
- [4] Avizienis, A., J. Laprie, and B. Randell. "Fundamental Concepts of Computer System Dependability." *IARP/IEEE-RAS Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments*, Seoul, Korea, May 2001.
- [5] Babaoglu, O. and A. Schiper. "On Group Communication in Large-Scale Distributed Systems." *ACM Operating Systems Review* 29(1):62-67, January 1995.
- [6] Backes, M. and C. Cachin. "Reliable Broadcast In A Computational Hybrid Model With Byzantine Faults, Crashes, And Recoveries" International Conference on Dependable Systems and Networks, San Francisco CA, June 2003
- [7] Birman, K. "The Process Group Approach to Reliable Distributed Computing." *Communications of the ACM*, 36(12):37-53 and 103, December 1993.

- [8] Burns, A., and A. J. Wellings. "Safety Kernels: Specification and Implementation." *High Integrity Systems* 1(3):287-300, 1995.
- [9] Carzaniga, A., D. Rosenblum, and A. Wolf. "Achieving Scalability and Expressiveness in an Internet-scale Event Notification Service." Symposium on Principles of Distributed Computing, 2000.
- [10] Cristian, F. "Understanding Fault-Tolerant Distributed Systems." *Communications of the ACM* 34(2):56-78, February 1991.
- [11] Deswarte, Y., N. Abghour, V. Nicomette, D. Powell. "An Intrusion-Tolerant Authorization Scheme for Internet Applications." Sup. to Proc. 2002 International Conference on Dependable Systems and Networks, Washington, D.C. June 2002.
- [12] Deutsch, M. S., and R. R. Willis. *Software Quality Engineering: A Total Technical and Management Approach*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- [13] Ellison, B., D. Fisher, R. Linger, H. Lipson, T. Longstaff, and N. Mead. "Survivable Network Systems: An Emerging Discipline." Technical Report CMU/SEI-97-TR-013, Software Engineering Institute, Carnegie Mellon University, November 1997.
- [14] Federal Aviation Administration Advisory Circular 25.1309-1A, "System Design and Analysis."
- [15] Fraser, T., L. Badger, and M. Feldman. "Hardening COTS Software with Generic Software Wrappers." in OASIS: Foundations of Intrusion Tolerant Systems (J. Lala Ed.), IEEE Computer Society Press, 2003.
- [16] Gartner, Felix C. "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments." *ACM Computing Surveys* 31(1):1-26, March 1999.
- [17] Jalote, P. *Fault Tolerance in Distributed Systems*. Prentice Hall:Englewood Cliffs, NJ, 1994.
- [18] Knight, J., M. Elder, J. Flinn, and P. Marx. "Summaries of Four Critical Infrastructure Systems." Technical Report CS-97-27, Department of Computer Science, University of Virginia, November 1997.
- [19] Knight, J. C., D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, P. Devanbu, and M. Gertz. "The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications." Intrusion Tolerance Workshop, The International Conference on Dependable Systems and Networks, Washington, DC, June 2002.
- [20] Knight, J. C., E. A. Strunk and K. J. Sullivan. "Towards a Rigorous Definition of Information System Survivability." DISCEX 2003, Washington, DC, April 2003.
- [21] Lala, J. "Foundations of Intrusion Tolerant Systems." IEEE Computer Society Press, Catalog # PR02057, 2003.
- [22] Leveson, N., T. Shimeall, J. Stolzy and J. Thomas. "Design for Safe Software." AIAA Space Sciences Meeting, Reno, Nevada, 1983.
- [23] Magee, J., N. Dulay and J. Kramer. "Structuring Parallel and Distributed Programs." *Software Engineering Journal*, 8(2):73-82, March 1993.
- [24] Magee, J., and J. Kramer. "Darwin: An Architectural Description Language." <http://www-dse.doc.ic.ac.uk/research/darwin/darwin.html>, 1998.

- [25] Melliar-Smith, P., and L. Moser. "Surviving Network Partitioning." *IEEE Computer* 31(3):62-68, March 1998.
- [26] Myers, J.F. "On Evaluating The Performability Of Degradable Computing Systems." *IEEE Transactions on Computers* 29(8):720-731, August 1980.
- [27] Myers, J.F., and W.H. Sanders. "Specification And Construction Of Performability Models." *Proc. Second International Workshop on Performability Modeling of Computer and Communication Systems*, Mont Saint-Michel, France, June 1993.
- [28] Nace, W., and P. Koopman. "A Product Family Based Approach to Graceful Degradation." *DIPES 2000*, Paderborn, Germany, October 2000.
- [29] Peters, D. K., and D. L. Parnas. "Requirements-based Monitors for Real-time Systems." *IEEE Trans. on Software Engineering* 28(2):146-158, Feb. 2002.
- [30] Powell, D. and R. Stroud (Eds). "Conceptual Model and Architecture of MAF-TIA." <http://www.newcastle.research.ec.org/maftia/deliverables/D21.pdf>
- [31] Ramasamy, H., P. Pandey, J. Lyons, M. Cukier, and W. Sanders. "Quantifying the Cost of Providing Intrusion Tolerance in Group Communications." in *OASIS: Foundations of Intrusion Tolerant Systems* (J. Lala Ed.), IEEE Computer Society Press, 2003.
- [32] Reynolds, J., J. Just, E. Lawson, L. Clough, R. Maglich, and K. Levitt. "The Design and Implementation of an Intrusion Tolerant System." in *OASIS: Foundations of Intrusion Tolerant Systems* (J. Lala Ed.), IEEE Computer Society Press, 2003.
- [33] Rowanhill, Jonathan C., Philip E. Varner and John C. Knight. "Efficient Hierarchic Management For Reconfiguration of Networked Information Systems." *The International Conference on Dependable Systems and Networks (DSN-2004)*, Florence, Italy, June 2004.
- [34] Rushby, J. "Kernels for Safety?" *Safe and Secure Computing Systems*, T. Anderson Ed., Blackwell Scientific Publications, 1989.
- [35] Sames, D., B. Matt, B. Niebuhr, G. Tally, B. Whitmore, and D. Bakken. "Developing a Heterogeneous Intrusion Tolerant CORBA Systems." in *OASIS: Foundations of Intrusion Tolerant Systems* (J. Lala Ed.), IEEE Computer Society Press, 2003.
- [36] Scandariato, Riccardo and John C. Knight. "An Automated Defense System to Counter Internet Worms." *Technical Report CS-2004-12*, Department of Computer Science, University of Virginia, March 2004.
- [37] Schlichting, R. D., and F. B. Schneider. "Fail-stop processors: An approach to designing fault-tolerant computing systems." *ACM Transactions on Computing Systems* 1(3):222-238.
- [38] Sha, L. "Using Simplicity to Control Complexity." *IEEE Software* 18(4):20-28, 2001.
- [39] Sha, L., R. Rajkumar and M. Gagliardi. "A Software Architecture for Dependable and Evolvable Industrial Computing Systems." *Technical Report CMU/SEI-95-TR-005*, Software Engineering Institute, Carnegie Mellon University, 1995.

- [40] Shelton, C., P. Koopman, and W. Nace. "A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems." Eighth IEEE International Workshop on Object-oriented Real-time Dependable Systems, Guadalajara, Mexico, January 2003.
- [41] Storey, N. *Safety-Critical Computer Systems*. Prentice Hall: Harlow, U.K., 1996.
- [42] Strunk, E. *The Role of Natural Language in a Software Product*. M.S. Thesis, University of Virginia Dept. of Computer Science, May 2002.
- [43] Strunk, E. A., and J. C. Knight. "Assured Reconfiguration of Embedded Real-Time Software." The International Conference on Dependable Systems and Networks (DSN-2004), Florence, Italy, June 2004.
- [44] Strunk, J., G. Goodson, M. Scheinholz, C. Soules and G Ganger. "Self Securing Storage: Protecting Data in Compromised Systems." in OASIS: Foundations of Intrusion Tolerant Systems (J. Lala Ed.), IEEE Computer Society Press, 2003.
- [45] Sullivan, K., J. Knight, X. Du, and S. Geist. "Information Survivability Control Systems." *Proc. 21st International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, May 1999.
- [46] U.S. Department of Commerce, National Telecommunications and Information Administration, Institute for Telecommunications Services, Federal Std. 1037C.
- [47] Valetto, G. and G. Kaiser. "Using Process Technology to Control and Coordinate Software Adaptation." 25th International Conference on Software Engineering. Portland, Or. May, 2003.
- [48] van Renesse, R., K. Birman, and S. Maffei. "Horus: A Flexible Group Communications System." *Comm. of the ACM* 39(4):76-83, April 1996.
- [49] van Renesse, R., K. Birman, M. Hayden, A. Vaysburd, and D. Karr. "Building Adaptive Systems Using Ensemble." Technical Report TR97-1638, Department of Computer Science, Cornell University, July 1997.
- [50] Van Renesse, R., K. Birman and W. Vogels. "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining." *ACM Transactions on Computer Systems*, Vol. 21, No. 2, pp. 164–206, May 2003.
- [51] Verissimo, P., Neves, N.F., and Correia, M. "Intrusion-Tolerant Architectures: Concepts and Design (extended)." Technical Report DI/FCUL TR03-5, Department of Computer Science, University of Lisboa, 2003
- [52] Wang, F., F. Jou, F. Gong, C. Sargor, K. Goseva-Popstojanova, and K. Trivedi. "SITAR: A Scalable Intrusion-Tolerant Architecture for Distributed Services." in OASIS: Foundations of Intrusion Tolerant Systems (J. Lala Ed.), IEEE Computer Society Press, 2003.
- [53] Wika, K.J., and J.C. Knight. "On The Enforcement of Software Safety Policies." *Proceedings of the Tenth Annual Conference on Computer Assurance (COM-PASS)*, Gaithersburg, MD, 1995.
- [54] Zhou, L., F. Schneider and R. Renesse. "COCA: A Secure Distributed Online Certification Authority." in OASIS: Foundations of Intrusion Tolerant Systems (J. Lala Ed.), IEEE Computer Society Press, 2003.