The Architecture of a Reliable Software Monitoring System for Embedded Software Systems

John Munson, Axel Krings, Robert Hiromoto
Computer Science Department
University of Idaho
Moscow, ID 83833-1010
{jmunson, krings, hiromoto}@cs.uidaho.edu

Abstract -- *We develop the notion of a measurement-based methodology for embedded software systems to ensure properties of reliability, survivability and security, not only under benign faults but under malicious and hazardous conditions as well. The driving force is the need to develop a dynamic run-time monitoring system for use in these embedded mission critical systems. These systems must run reliably, must be secure and they must fail gracefully. That is, they must continue operating in the face of the departures from their nominal operating scenarios, the failure of one or more system components due to normal hardware and software faults, as well as malicious acts. To insure the integrity of embedded software systems, the activity of these systems must be monitored as they operate. For each of these systems, it is possible to establish a very succinct representation of nominal system activity. Furthermore, it is possible to detect departures from the nominal operating scenario in a timely fashion. Such departure may be due to various circumstances, e.g., an assault from an outside agent, thus forcing the system to operate in an off-nominal environment for which it was neither tested nor certified, or a hardware/software component that has ceased to operate in a nominal fashion.*

A well-designed system will have the property of graceful degradation. It must continue to run even though some of the functionality may have been lost. This involves the intelligent remapping of system functions. Those functions that are impacted by the failure of a system component must be identified and isolated. Thus, a system must be designed so that its basic operations may be remapped onto system components still operational. That is, the mission objectives of the software must be reassessed in terms of the current operational capabilities of the software system. By integrating the mechanisms to support observation and detection directly into the design methodology, we propose to shift away from the currently applied paradigm of addressing reliability, security and survivability in an add-on fashion at the end of the software development process. Rather, the integrity monitoring ability will be integrated into the overall architecture of the software system. The measurement and control methodology developed under this research program will readily migrate into hardware, leading to the development of new hardware architecture with built-in survivability, security and reliability attributes.

I. INTRODUCTION

The principal objective of this paper is to articulate a methodology for the design of high-confidence embedded systems. These systems must run reliably, must be secure and must fail gracefully. They must continue operating in the face of a failure of one or more system components, may it result from normal hardware and software faults as well as malicious act like hacking, virus or Trojans. Embedded systems integrate both hardware and software components that form a self-consistent system that is expected to function without human intervention. The design challenge is to develop systems that are self-aware and capable of monitoring their components during program execution. Our approach to the design of mission-critical embedded systems that draws upon the advances made in the four disciplines of software measurements, survivable systems, system security, and reconfigurable system architectures. A hierarchical embedded system approach will be outline to address the functional topological organization of the system to withstand adverse interruptions; to remap essential system's activities for survivability; to detect damages or failures within a sensor-networking framework; and to develop flexible software/hardware reconfiguration policies.

There are three principle aspects of fault events that mitigate the development of high-confidence embedded systems.

First, a failure event in the operation of embedded systems must be *observable*. That is, any operational process that deviates from its certified sequence of execution must be observable so that an exception can be raised. This departure will, of necessity, first be observed in one of the program modules that constitute the execution primitives of the operation. At the lowest level each system is comprised of primitive modules, e.g., code modules. These modules are structured in the design process to implement basic, elemental, system operations. Thus, the topology of a system must be designed to reflect this organization.

Second, a failure event that occurs in a specific code module must be *detectable* at the point that it occurs. The detection of the failure event is central to this discussion, as it will have direct implications with respect to the need for remapping techniques. The first step in this process is to suitably instrument a system with sufficient software and hardware probes to capture the failure event and to initiate the failure analysis process [13]. The second step of the detection process is the notion of damage assessment. The failure of a function implemented by a particular code module may well propagate to other functions that are closely related to the failed module in the system topology.

Finally, the system under failure events must be *recoverable*. It is one thing for a system component to fail. It is quite another for this failure event to bring the whole system down. One characteristic of a high-confidence embedded system must be the property of *graceful* degradation. That is, it must continue to run even though some of the functionality may have been lost. This involves the intelligent remapping of system functions or modules. Those functions that are impacted by the failure of a system component must be identified and isolated. In this sense, a system must be designed so that the basic operations of the system may be remapped onto system components that are still operational. That is, the mission objectives of the embedded software must be reassessed in terms of the current operational capabilities of the software system [5].

The first key contribution of our new approach is the characterization of the reliability of an embedded system in terms of the system's certified activities as it executes its various operations and its implication on system survivability. The reliability assessment of the program will be accomplished dynamically while the program is executing to identify, with reasonable precision, which software component are unreliable and how these components can be remapped to auxiliary components. In other words, the system will be instrumented to appraise its own health in order to engage principles of survivability. It is understood that no software system can be thoroughly or exhaustively tested. However, it is possible to certify a range of software behaviors that represent the certified operational profiles of a correct, design-specified embedded system. Taking these certified operational profiles as the standards for measuring the reliability of the embedded system, the direct monitoring and identification of all unusual or abnormal software activities can be treated as indications of a potentially compromised system. The status of the system under these abnormal conditions can then be analyzed and recertified as either acceptable or unacceptable system behavior.

The second key contribution is the exploitation of the reliability model to achieve system survivability. Specifically, the reliability model is especially suitable to overcome the limitations of the non-formal survivability definitions in [5]. Rather, we propose survivability approaches that adopti the formal *survivability definitions* [15] and leveraging [9]. This expands other approaches taken shifting the survivability considerations from the survivability Markov models to real-time survivability [8].

By incorporating these capabilities directly into the system design methodology, we drastically shift away from the current paradigm that addresses reliability, security and survivability in an add-on fashion, occurring at the end of the design cycle. Instead a unified and integrated design methodology for embedded systems is outlined below.

II. MOTIVATION

Existing approaches applied to system reliability either do not differentiate between software and hardware failure events or apply models developed mainly from a hardware reliability perspective [14]. Hardware components have the advantage that direct monitoring of their proper operation can be observed and detected at the component level. Software failure events, on the other hand, are handicapped by the false assumption that software failure events are observable. Furthermore, it is assumed that it is possible to measure with some degree of accuracy the time between these failure events or the actual (real-time clock) time when the failure occurred. The simplest example of this improbability of measuring the time between failures of a program may be found in a program that hangs in an infinite loop, which has real implications with respect to recovery. Technically the failure event happened on entry to the loop. The program, however, will continue to execute until this behavior is detected. This may take seconds, minutes, or hours depending on the patience and/or attentiveness of the operator. The fact is that the overwhelming majority of software failures go unnoticed when they occur. Only when these failures disrupt the system by second, third or fourth order effects do they provide enough disruption for outside observation. Consequently, only those dramatic events that lead to the immediate collapse of a system can an observer see directly. The more insidious failures will lurk in the code for a long interval before their effects are observed. Failure events go unnoticed and undetected because the software has not been instrumented to detect these failure events. It is, therefore, presumptuous to attempt to model the reliability of software based on these inaccurate and erroneous data. The same argument can be made when considering error or fault propagation and fault-containment.

As long as users perform each system operation as prescribed by the intended system design, will each corresponding operation cause the system to select a pre-established, *certified* subset of its modules to execute. In this way, users tend to execute subsets of the total program operations. This has been observed and considered in the analysis of software specifications [17]. It is interesting to note, however, that two users of the same software may have totally different perceptions as to the reliability of the same system depending upon the mix of operations selected. This is directly attributable to the fact that these users are effectively running two different programs containing subsets of the entire program. Barring hardware failures, the failure of one or more of the modules in this subset can occur if either incorrect input is detected (data types, variable range, etc.) or a choice of non-standard combination of operations is selected. This is the principal reason that a program may have demonstrated great reliability in past performances but may suddenly become quite unreliable when required to adapt to changes in the manner in which the user interacts with the system. In other words, the past performance of a software system, from the standpoint of reliability, is not at all a good predictor of the future reliability of the system. Our previous investigations into the etiology of software failures in the Primary Avionics Software System of the Space Shuttle have provided substantial insight into this view of software reliability. On the basis of that work, it is now clear to us that it is not the software system that fails but rather the software system executing a particular operation that fails.

III. MODEL DESCRIPTION

One of the single most important aspects of the system's reliability is its behavior at run-time, i.e., its run-time profile.

III.A Certified Profile

Perhaps the greatest threat to the reliable operation of a modern embedded software system is the unanticipated operational demands placed on the system by the operational environment. As a consequence the system may well shift from a reliable (certified by the software developer) operational profile to an uncertified profile. In the event that a system is driven to operate under an uncertified operational profile, it will be important to understand whether the new behavior can be tested and possibly certified as reliable. This assumes, of course, that the software system is suitably instrumented to provide sufficient information to reconstruct the system activity and validate the correctness of that behavior and the associated system components. The main objective of the dynamic measurement methodology from the standpoint of high-confidence software systems is to trap, in real-time, any behavior that is considered uncertified. We hypothesize that it is possible from the observed behavior of the software modules to determine with a certain level of confidence the future reliability of a system under one or more certified operational profiles.

III.B Uncertified Profile

We subscribe to the notion that uncertified behavior constitutes anomalous behavior that has important consequences. The detection of an anomalous behavior is a likely indication that the software/hardware component has failed, a malicious attack has occurred, or that the users has initiated a sequence of uncertified operations [3]. The failure event itself is made tangible through such an anomalous process. The first step, then, in the design of a high-confidence system is the creation of a contingency management system to detect the anomalous activity of an executing system and resolve the problem at the lowest level of program module granularity.

III.C. General Design Overview

Given that a failure has occurred there are two possible avenues that the design process may take to insure that the software system will continue to operate after the failure event. First, the software system may be allowed to continue operating by eliminating the functionality that led to the failure event. Alternatively, the software may be designed to duplicate vital or mission critical functionalities by an associative remapping of available functionalities; in which case, the failure of a critical functionality can be recovered once detected. Dependability considerations are based on the concept that, to insure a high level of reliability in an executing software system, a program must be designed to work in conjunction with a real-time monitoring system. The principle considerations are two-fold. On the one hand, one has to monitor the activity of an executing program for the distribution of activity across program modules and also for potential failure events at the module level. The second consideration is to control the structure of the executing program. To perform this function, one needs to provide a new loaded program with an initial call structure that will control the execution sequence of program modules in the executing program. Furthermore, one may have to alter the structure of the executing program in the event that a failure has been detected at the program module level.

The general design methodology and system is displayed in Figure 1, which shows the executing program and the contingency management system. The blocks indicate a sequence of operations and core concepts and will be italicized as they are introduced in the following sections.

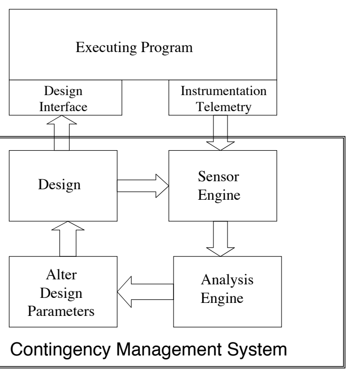

Fig. 1. General Design Overview

III.C.I Executing Program

In this design environment, the *Executing Program* has two distinct interfaces with the monitoring system. The *Design Interface* captures the structure of the call tree that represents the call sequences of the program modules as determined in the design process. The transfer of program control among all program modules occurs through this interface. This permits the program structure to be remapped during the program execution. As the program is executing, telemetry from the executing modules will be transmitted to the *Instrumentation Telemetry* section. This telemetry has two separate components. First, there is the record of transfer of control from one module to another. This permits the dynamic assessment of program activity (as in operational, functional, and module profiles). Also, failure events may be captured at the module level. The precise nature of the failure event together with the failed module name is also part of the basic telemetry package. The telemetry from the executing program is sent directly to the sensor engine in the dependability system. The results of the telemetry are then transferred to the analysis engine. There are two different types of program activity that the analysis engine processes. First, there is the flow of nominal program activity telemetry. Second, there are failure event data.

III.C.II Contingency Management System

In order to link the executing program with the contingency management system an information flow needs to be established that allows for the real-time monitoring and analysis of the executing program. To this end, as an integral part of the software design process (*Design*), a software system will be suitably instrumented so that its operation may be monitored when it is executing. It is quite reasonable, then, to integrate a *Sensor Engine* into the software operational scenario so that the operation of the system may be monitored as the software executes. As the system executes, sooner or later it may encounter a module with a fault in it. Execution of this fault may result in the failure of the software component that contains the fault. The sensor engine that monitors the execution of the software system may detect this failure event. In the event that a failure event does occur, it will be important to determined the nature of the failure event and ameliorate the effect of the failure so that the software system may continue to execute. There must be, then, some sort of analysis (*Analysis Engine*) that will occur after the detection of the failure event to determine a fall back strategy for the software system. This fallback strategy is articulated in the design parameters. The original *Design Parameters* that govern the software execution must be altered so that the system may resume executing. This will result in the expression of a new set of design parameters that will govern the operation of the system when it continues its execution.

IV. MODEL DEFINITIONS

To lay the foundation for a measurement-based, dynamic monitoring system that permits the real-time assessment of software reliability, it is necessary to establish a model for program execution that lends itself to a suitable instrumentation for the monitoring and failure analysis processes. In the subsequent discussion of program operation, it is useful to make the description of program specification, design and implementation somewhat more precise by introducing some notational conveniences. Let us begin this discussion by observing the fact that there are really two distinct abstract machines or models that define the implementation in the development of any software system.

IV.A. Operational and Functional Machine

The first software abstract machine is the *operational* machine. This is the machine that interfaces directly with the hardware interface. The embedded system provides a suite of services to the hardware system. Each of these services will cause the operational machine to perform a series of actions called *operations*. Each of these operations, in turn, causes the operational machine to perform some specific action. It is the purpose of this operational machine to articulate exactly *what* the software system must do to provide the necessary services dictated by the embedded software system requirements.
The second abstract machine is the *functional* machine. This machine is animated by a set of *functionalities* that describes exactly *how* each system operation is implemented. Whereas the operational abstract machine articulates what the software system will look like to the hardware system in which it is embedded, the functional abstract machine is that entity actually created by the software design process. We now turn our attention to the precise relationship between the operational abstract machine and the functional abstract machine. It is quite conceivable that we could construct a system wherein there is a one-to-one mapping between the user's operational model and the functional model. That is, for each user operation there is exactly one corresponding functionality. In most cases, however, there may be several discrete functionalities that must be executed to express the system services provided by the operational abstract machine.

IV.B. Definitions

Each operational machine consists of a set, O, of operations that animate it. Each functional system will have a set, F, of functionalities that animate it. For each operation, $o \in O$, that the system may perform, there will be a subset, $F^{(o)} \subseteq F$, of functionalities that will implement it. It is possible, then, to define a relation *IMPLEMENTS* over $O \times F$ such that IMPLEMENTS (o, f) is true if functionality f is used in the implementation of an operation, o. Within each operation one or more of the system's functionalities will be expressed. For a given operation, o, these expressed functionalities are those with the property $F^{(o)} = \{f : F \mid IMPLEMENTS(o, f)\}$.
Each functionality exercises a particular aspect of the functional machine. As long as the system's operational profile remains stable, the manner in which the functional machine actually executes is also stable. However, when there is a major shift in the operational profile by the system, then there will be a concomitant shift in the functional profile as well. This redistributes the activity of the functional machine and results in uncharacteristic behavior of the functional machine. This change in the usage of the system constitutes anomalous system activity. It should be noted that this definition of anomalous system activity is much more precise than that used in intrusion detection, i.e., anomaly detection [18].

Let M be the set of program modules of a system. The software design process is then basically a matter of assigning functionalities $f \in F$ to specific program modules $m \in M$. The design process may be thought of as the process of defining a set of relations, *ASSIGNS,* over $F \times M$ such that *ASSIGNS*(*f, m*) is true if functionality f is expressed in module m.

Each operation in O is distinctly expressed by a set of functionalities. If a particular operation, o, is defined by functionalities f_a and f_b, then the set of program modules that are bound to operation o is $M^{(o)} = M^{(f_a)} \cup M^{(f_b)}$, where $M^{(f_a)}$ and $M^{(f_b)}$ represent the set of program modules associated with functionality f_a and f_b. In general, $M^{(o)} = \bigcup_{IMPLEMENTS(o,fi)} M^{(f_i)}$.

IV.C. Mappings

There is a distinct mapping from the set of operations to the set of program modules. Each operation is associated with a distinct set of functionalities. These individual functionalities are, in turn, associated with a distinct set of modules. The mapping will be explained using the example shown in Figure 2. In this figure we can see that there are three operations (o_1 through o_3) that map into a set of five functionalities (f_1 through f_5) that are implemented, in turn, by a total of eight program modules (m_1 through m_8). It is clear from this figure that if the system exercises the operation o_3 substantially, then modules m_6, m_7 and m_8 will also be heavily used.

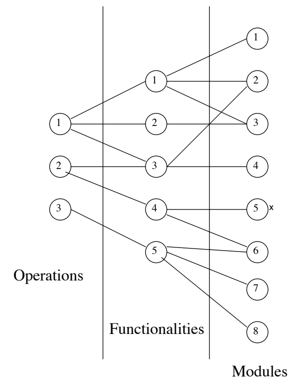

Fig. 2. Mappings

V. SYSTEM BEHAVIOR

A significant problem incurred in the development of embedded systems is the simple fact that there is no concept of time in a software system. Programs are at once ethereal and eternal. There is no real concept of time in the physics of software systems. The notion of time must be imposed from the outside world perhaps in the form of a real-time clock interrupt or some other such mechanism. For our purposes, we will measure the progress of a system executing a sequence of operations or functionalities or modules in terms of *epochs* [13]. An *operational epoch* is the transition from one operation to another. Similarly, a *functional epoch* is the transition from one functionality to another. Finally, a *module epoch* is the transition from one module to another.
The behavior of the system will next be considered with respect to its user operational profile, functional profile and module profile, which we denote by u, q and p respectively

V.A. Operational Profile

The operational capabilities of an embedded software system are specified during the system requirements specification phase. The end product of the system requirements analysis is a set of specifications that articulates the nature of the operations that the system performs. Each of the hardware systems that contain the embedded software system will exercise its embedded software system with a specific subset of these operations. In this process, the system typically will not use all of the operations with the same probability. The operational profile of the software system is defined as the set of unconditional probabilities of each of the operations, *o*, being executed by the hardware.

One approach to quantifying the mix of user operations selected for execution is to use a stochastic process to describe module epochs, i.e., the transition of program modules from one to another as a user transitions among operations. Let *Z* be a random variable defined on the indices of the operations in *O*. Then, $u_l = \Pr[Z = l], l = 1,2,\ldots,|O|$ is the probability that the system is executing an operation o_l as specified in the operational specification of the program and $|O|$ is the cardinality of the set of operations. We can now define the *operational profile* formally as a vector $\mathbf{u} = < u_1, u_2, \ldots, u_{|O|} >$ representing the distribution of activity or usage among the set of operations. The distribution of the operational profile is multinomial for programs designed to fulfill more than two distinct operations.

Given the above foundation for describing the distribution of activity of a system across a set of distinct operations, it is possible to define more precisely the notion of the *behavior* of the system in regards to the embedded software system. This requires measuring the transition of the system from one operation to another as an operational epoch. If we observe the sequence of operations performed by the system over time, we may tally the frequency-of-use, or spectrum *s*, of each operation. We may derive an estimate for the distribution of system activity, i.e., usage, denoted by $\hat{u}_i$, on a particular operation o_i after *n* operational epochs. Specifically, over an interval of *n* epochs during which o_i was used s_i times, we get $\hat{u}_i = s_i / n$. Note that the sum of all s_i in *n* epochs is maximally *n*. We will refer to s_i as the frequency of o_i. The *observed operational profile* for this interval of *n* epochs is represented by a n-tuple $\hat{\mathbf{u}} = < \hat{u}_1, \hat{u}_2, \ldots, \hat{u}_{|O|} >$. Thus, $\hat{\mathbf{u}}$ indicates the fraction of time specific operations were used over *n* operations, i.e., over *n* epochs. The observed profile, $\hat{\mathbf{u}}$, can serve as an estimate for the user operation profile **u**. It is clear that this estimate, $\hat{\mathbf{u}}$, for the user operation profile is a point in the *n* dimensional space of the operational profile.

Now that we have defined the observed operational profile $\hat{\mathbf{u}}$ with respect to *n* epochs, let's consider sequences of operational profiles, each consisting of *n* operations. Let $\hat{\mathbf{u}}^k$ denote the k^{th} operational profile. Again, each of the vectors $\hat{\mathbf{u}}^k$ represents a point in the *n-1* dimensional behavior space of the user. These individual profiles, or usage patterns, are clustered about a centroid, $\overline{\mathbf{u}}$, in the *n-1* dimensional behavior space. Centroid $\overline{\mathbf{u}}$ is defined over *m* sequences of *n* epochs. Specifically, the elements $\overline{u}_i$ of $\overline{\mathbf{u}}$ are defined as $\overline{u}_i = \frac{1}{m} \sum_{j=1}^{m} \hat{u}_i^j$. Each of the observed operational profiles $\hat{\mathbf{u}}^k$ is within a specific distance d_k of the centroid, where $d_k = \sum_{i=1}^{n} (\overline{u}_i - \hat{u}_i^k)^2$. One may define an arbitrary distance ε about the centroid so that all points in the observed operational profiles are included. This ε *neighborhood* defines the behavior of the system over which $\overline{\mathbf{u}}$ is defined, i.e., the period of observation.

V.B. Functional Profile

As a developing software system enters the high-level design stage, the operational model must be mapped onto a functional model that serves as the foundation for the coding process. From this high-level design process, the precise nature of the functional machine will be developed. Each operation of the operational machine will be implemented by one or more functionalities that drive the function/design model. The distribution of activity among the various system functionalities will be represented by the system *functional profile*. The functional profile of the software system is the set of unconditional probabilities for each functionality in *F* executed by the system. Let ***Y*** be a random variable defined on the indices of the functionalities in *F*. Similar to the operational profile in the previous section we can now define a functional profile as $\mathbf{q} = < q_1, q_2, \ldots, q_{|F|} >$, where $q_k = \Pr[Y = k],\ k = 1, 2, \ldots, |F|$ is the probability that the system is executing program functionality f_k as specified in the functional requirements of the program. A program executing on a serial machine can only be executing one functionality at a time. The distribution of ***q***, then, is multinomial for programs with multiple functionalities. The probabilities q_k are dependent on how the system distributes its time across the suite of system operations. Given functionality f_k and operation o_l we can now define the probability that f_k is used by o_l as $w_{kl} = \Pr[Y = k \mid Z = l]$. That is, if we know the particular operation being performed, then we can determine the distribution of activity among the various functionalities.

The joint probability that a given operation o_l is being expressed and the system is executing a particular functionality f_k is given by $\Pr[Y = k \cap Z = l] = \Pr[Z = l]\Pr[Y = k \mid Z = l] = u_l w_{kl}$. Thus, the unconditional probability q_k of executing functionality f_k under a particular operational profile is $q_k = \Pr[Y = k] = \sum_l \Pr[Y = k \cap Z = l] = \sum_l u_l w_{kl}$.

V.C. Module Profile

The manner in which a program exercises its many modules as the system executes a sequence of operations of the embedded software system is determined directly by the design of the program. Certain subsets of modules are employed to implement each of the functionalities. Each time that a functionality is invoked in the implementation of an operation, a distinct subset of program modules will execute. The distribution of activity of program modules depends on the usage of functionalities.

Analogous to the definitions expressing the relationship between operations and functionalities we can express the *module profile* as $\mathbf{p} = < p_1, p_2, \ldots, p_{|M|} >$. The conditional and unconditional probabilities with respect to the occurrence of modules follow directly.

VI. MODEL ANALYSIS

We now explore how a program is structured as it executes each of the user operations. As a program executes a particular user operation, it transfers control from one module to another. There is always a main program module that receives control as the program begins to execute. The structure of the executing program may be represented as a call tree. Each program functionality is represented by one or more sub-trees of this call tree, depending on the number of operations that are implemented by that functionality.

VI.A. Operation Dependence Graph

A *demand-driven call-tree* (DDCT) is constructed for each certified operation. Memorize the dynamic user operation applications into an operation-lookup (OLU) table for future analysis. The DDCT is a directed graph defined with an underlying support algebra that provides a convenient venue for applications of formal analysis. The DDCT compilations define variable usage at the module lever as well as a call tree that uniquely specifies each certified user operation. The OLU table contains the subset of program modules and calls that actually occur when the program is executing under a particular user operational profile. However, a second call tree that can be constructed by formal analysis is what we term the *feasible call-tree*. This tree is comprised of nodes representing all possible program modules and arcs that represent all possible module call combinations. In practice, elements of poor program design, such as the use of function pointers, may preclude the exact determination of the feasible call tree.

Within the context of the DDCT, specific functionalities may be identified for use in implementing other functionalities. Beginning with the main program module, a master-level functionality for the entire program exits. In one sense, all other program functionalities implement this master-level functionality. Each functionality has a module at the root of its sub-tree that is unique to that functionality. This means that nodes in that sub-tree are not elements of the next upper level functionality. Thus, the elements of the sub-tree headed by the root node module of functionality f_b are not necessarily also elements of functionality f_a.

Program modules are structured in terms of calls among the program modules into call trees. Functionalities are expressed as sub-trees of call trees. There is a distinct feasible call sub-tree for each functionality. A functionality sub-tree is constructed from a call sub-tree by substitution. A functionality node replaces each call sub-tree representing that functionality in the original call tree. The end result of this substitution is the construction of a functionality sub-tree displaying the interrelationship of the set of possible functionalities. This reduction may be carried yet one further step. Since each operation is comprised of functionalities that are organized into functionality sub-trees, substituting an operational node for the functionality sub-trees that comprise that node can reduce the functionality tree. In this final step the operational tree is created revealing the operational structure of the program.

Once the functional analysis is completed, the demand-driven-call-tree can be inverted to examine its data-flow behavior, thus exposing the effects of fault propagation under consideration of pathological behavior [7]. Under this analysis additional rewrite rules can be applied to expose execution parallelism [1].

VI.B. Reliability, Security and Survivability

The most important consideration, from the standpoint of system survivability, security and reliability, is the mapping from the set of user operations to the set of program modules. A software failure event can occur at the program module level. The failed program module may be associated with one or more operations in the operational model [3]. This implies that a particular failure event at the module level results in the failure of one or more user operations. The failure of a software system, then, is dependent only on what the software is currently doing: the operations that a user is performing. If a program is currently executing an operation that is expressed in terms of a set of fault free modules, this operation can certainly execute indefinitely without any likelihood of failure [11, 12]. A failure event can only occur when the software system executes a module that contains faults. If an operation is never selected that drives the program into a module that contains faults, then the program will never fail. Alternatively, a program may well execute successfully in a module that contains faults just as long as the faults are in code subsets that are not executed [10].

The distribution of faults in a software system is most decidedly not uniform. Faults are located in program modules in relationship to certain well-defined program attributes [12]. Some program modules are quite fault prone while others are relatively fault free. Clearly, some operations might well invoke a set of fault prone modules while other operations will tend to use modules that are relatively fault free. The distribution of program activity across the set of modules potentially containing faults is a very important piece of information.

By keeping track of the state transitions from module to module and operation to operation we may come to understand the relative fault exposure of various execution scenarios. This information coupled with the operational profile tells us just how reliable the system is when it is deployed. Programs make transitions from module to module as they execute. These transitions may be observed. Transitions to program modules that are discovered to be fault-laden results in an increased probability of failure. We can model these transitions as a stochastic process. Ultimately, by developing a mathematical description for the behavior of the software as it transitions from one module to another driven by the operations that it is performing, we can describe the operational reliability of the program. If we can know the reliability of the operations and how the system apportions its time among these operations, we can then know the reliability of the system. In a sense, this constitutes a real-time usage pattern based reliability model that is much more powerful than the model described in [17].

From this new perspective, software survivability may be achieved through the understanding of software activity in one of two distinct ways. First, a thorough analysis of the operational reliability of a system must be established before the system is actually deployed. Secondly, these systems are characterized by functionalities that are critical to their mission. Through the application of survivability techniques, the failure of one system component may be averted from a total catastrophic collapse of the software system. The survivability considerations are met though a partition of the system functionalities into vital and non-vital components.

In Figure 3 we see a hypothetical sequence of execution of several functionalities. Functionality f_a is followed by functionality f_b which is followed by functionality f_c and f_d. From a reliability point of view this constitutes a acyclical reliability block diagram [14]. At the design stage we may classify each functionality as to its relative importance to insuring the meaningful operation of the software system. In this process we might, for example, determine that functionality f_b is a vital functionality. To insure the reliability of this system component we may chose to implement the functionality in two distinct functionalities, f_{b1} and f_{b2}. These functionalities are essentially identical. They are implemented, however, by two disparate sets of program modules. These functionalities are always run in parallel, say with two different execution threads. If there is a failure in
one program module that implements a functionality, then that execution thread will be abandoned. The result is a parallel reliability block diagram, i.e. an 1-of-n configuration. In this way, reliability of a system can be manipulated by taking advantage of parallel reliability blocks and, given the dynamic derivation of the reliability model, a run-time reliability optimization could be achieved extending the ideas in [2].

In Figure 3, we also can see that there is an execution path around functionality f_d. In this case, this particular functionality is not vital to the continuing operation of the system. If there is a failure in one of the modules that implement this functionality, the subsequent execution sequences simply bypass the functionality. In this case, there is a loss of system functionality, but the system can continue to operate in a depredated fashion.

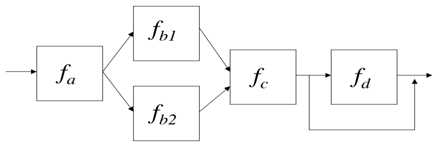

Fig. 3. Execution sequence

We now return the discussion to the analysis engine that was shown in Figure 1 to consider its role when the program is executing in a nominal fashion. We are aware that a program that has been certified for certain behaviors executes in a reliable fashion. The analytical engine's role here is a fairly simple one. A program is executing in a nominal fashion if it may be determined that the profiles (operational, functional, and module) are within a specific tolerance factor. We may determine this by noting that a profile, say module profile, represents a point in an n dimensional space. During a particular observation interval we may measure the distance between the certified module profile $\boldsymbol{p}^c$ and the most recently observed module profile, $\boldsymbol{p}^o$, as $d = \sum_i (p_i^c - p_i^o)^2$. A system may be deemed to be operating in a nominal or pre-certified fashion if $d \le \varepsilon$, where ε is a pre-established threshold for nominal program activity. If a program is found to be operating in an off-nominal fashion, the modules that are involved in the new behavior are those with the property $p_i^c - p_i^o < 0$. The subset of modules for which this inequality holds may easily be mapped to a specific functionality and thence to one or more operations. This change in behavior may then be employed to recertify the new operation profile of the system.

In the event of a failure in a program module, it is necessary to identify the particular functionality that was being expressed at the point of the failure. Depending on the design there are several options open to continue the program execution. First, let us observe that the failed module is an element of a particular functionality. That functionality is, in turn, used to implement one or more operations. If the operations in question are non-vital operations, we may simply remap the program execution tree to exclude the failed operations and return the program to its new restricted operational status. It is quite possible that the failed module is determined to be part of a vital functionality/operation. If this is the case, then it is not possible to simply map around the faulty functionality. We must implement a reliability strategy long employed in the design of mission-critical hardware systems. We implement each critical functionality into multiple distinct call sub-trees, as dictated by the fault model we want to select. This means that all functionalities that are determined to be vital run in a redundant fashion. Each of the call sub-trees will be implemented with different sets of modules at the design stage. These redundant functionalities run in parallel (possibly as execution threads) when the program requires that functionality. In this case, the thread containing the failed program module may simply be prevented from executing in the future.

Finally, it should be pointed out that security is an immediate consequence of our software approach. System security is treated as a general fault scenario that obviates the need to implement traps or guards into the software/hardware specifications to counter specific attacks; making our approach robust.

VII. CONCLUSIONS

The ultimate objective of our research is to develop a rigorous methodology for the design and implementation of embedded software system. The driving force behind this research is the development of a disciplined engineering approach to this software development. Central to this development process is the empirically based design and implementation strategies. This project provides the foundation toolset for the necessary measurement activities for these design and implementation. It also provides the foundation research methodology for the conduct of inquiry into the development of engineered software systems.

REFERENCES

[1] Arvind and X. Shen. *Using Term Rewriting Systems to Design and Verify Processors*, IEEE Micro Special Issue on Modeling and Validation of Microprocessors, May/June 1999.

[2] I. Assayad, A. Girault, and H. Kalla. *A bi-criteria scheduling heuristics for distributed embedded systems under reliability and real-time constraints*. In International Conference on Dependable Systems and Networks, DSN'04, Firenze, Italy, June 2004.

[3] S. G. Elbaum and J. C. Munson, "Intrusion Detection through Dynamic Software Measurement", *Proceedings of the USENIX Workshop on Intrusion Detection and Network Monitoring ,* Santa Clara, CA, April 1999.

[4] S. G. Elbaum and J. C. Munson, "Software Black Box: An Alternative Mechanism for Failure Analysis", *Proceedings of the 2000 IEEE International Symposium of Software Reliability Engineering ,* San Jose, CA, November 2000.

[5] R. Ellison, D. Fisher, H. Lipson, T. Longstaff, and N. Mead, *Survivability: Protecting Your Critical Systems*, IEEE INTERNET COMPUTING, Vol. 3, No. 6; NOVEMBER-DECEMBER 1999, pp. 55-63.

[6] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. *The MAFT architecture for distributed fault tolerance*. IEEE Transactions on Computers, 37(4):398--405, April 1988.

[7] Krings Axel W., Jean-Louis Roch, and Samir Jafar, "*Certification of Large Distributed Computations with Task Dependencies in Hostile Environments*", IEEE Electro/Information Technology Conference , (EIT 2005), May 22-25, Lincoln, Nebraska, 2005

[8] Yun Liu and Kishor S. Trivedi, *A General Framework for Network Survivability Quantification*, in Proceedings of the 12th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB) together with 3rd Polish-German Teletraffic Symposium (PGTS), Dresden, Germany, September 2004.

[9] S.C. Liew and K.W. Lu. *A framework for characterizing disaster-based network survivability. IEEE Journal on Selected Areas in Communications*, 12(1):52–58, January 1994.

[10] J. C. Munson, "A Functional Approach to Software Reliability Modeling," in Boisvert, ed., Quality of Numerical Software, Assessment and Enhancement, Chapman & Hall, London, 1997. ISBN 0-412-80530-8.

[11]J. C. Munson and S. G. Elbaum, "Software Reliability as a Function of User Execution Patterns", *Proceedings of HICSS-32*, Hawaii, 1999.

[12] J. C. Munson and A. P. Nikora, "Toward a Quantifiable Definition of Software Faults," *Proceedings of the 2002 IEEE International Symposium of Software Reliability Engineering*, Annapolis, MD, December 2002.

[13]J. C. Munson, *Software Engineering Measurement*, CRC Press, 2003, ISBN: 0849315034

[14] Robin A. Sahner, Kishor S. Trivedi and Antonio Puliafito, *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package,* (The Red Book), Kluwer Academic Publishers, 1996.

[15] Working Group on Network Survivability Performance. *Technical report on enhanced network survivability performance*, Technical report, February 2001.

[16] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock. *SIFT: Design and analysis of a fault-tolerant computer for aircraft control*. Proceedings of the IEEE, 66(10):1240--1255, 1978.

[17] Whittaker James A., and J.H. Poore, *Markov Analysis of Software Specifications*, ACM Transactions on Software Engineering and Methodology, Vol.2, No.1, January 1993, pp. 93-106.

[18] Allen, et.al., *State of the Practice of Intrusion Detection Technologies*, Technical Report , CMU/SEI-99-TR-028, ESC-99-028, January 2000.