Supporting the Evolution of Distributed, Non-stop, Mission and Safety Critical Systems

Charles W. McKay and Colin Atkinson
University of Houston - Clear Lake
2700 Bay Area Boulevard, Houston, TX. 77058.
Phone: +713 283 3830, Fax: +713 283 3869
E-mail: mckay@cl.uh.edu

Keywords: distribution, environments, non-stop, real-time, safety-critical

Edited by: Marcin Paprzycki and Janusz Zalewski
Received: February 12, 1994 **Revised:** October 20, 1994 **Accepted:** January 9, 1995

In coming years embedded systems which are distributed, non-stop and "mission and safety critical" (MASC) are likely to assume increasing importance. The construction, operation and maintenance of this class of system presents a unique blend of problems which many traditional tools and techniques, targeted to just one problem area, cannot currently address. This paper provides an overview of a promising, model-based framework for supporting such systems that has been developed as part of NASA's MISSION project. Based on well-established research advances in computing, the MISSION approach provides a domain-specific, life-cycle support framework encompassing three separate environments: host, integration and target. Although the individual elements of the framework are not all new, their synergistic packaging within the MISSION project is believed to be unique. This paper focuses upon the systems-level support for applications executing in the target environment.

1 Introduction

An embedded system is a computer system which is constructed to monitor and/or control a set of devices and processes constituting some larger engineering system. The term "embedded" is used to reflect the fact that such computing systems are physically encapsulated by the engineering system they monitor/control. An important characteristic of embedded systems is that they are typically real-time - not only must they produce the correct result, but they must do so within a specified period of time. Because of their monitoring and controlling role, the reliable execution of an embedded system is often critical to the success of the overall mission and to the safety of life, health, property or the environment. In such circumstances the embedded system is termed a mission and safety critical (MASC) system.

As the reliability and efficiency of networking technology has increased, and the cost of microprocessors has plummeted, there has been an increasing trend towards the implementation of embedded systems as distributed systems made up of autonomous, cooperative processors interconnected by communication channels. Not only does such an implementation enable processing power to be located physically close to the individual devices in the system, but it also opens up the possibility of extending, or modifying, parts of a system while other parts are still running. In other words, it opens up the possibility of building non-stop systems which can be dynamically upgraded and reconfigured.

In coming years there is likely to be an increasing need for embedded systems which exhibit all the properties identified above, namely the properties of being mission and safety critical, real-time, distributed and non-stop. Such systems are essential in extremely hostile and/or inaccessible environments, such as space or the depths of the ocean, and are therefore crucial to pending NASA projects (e.g., space station, lunar outpost, human missions to Mars). Such systems are also

likely to be used in large process control applications such as factory automation, power plant control, etc.

In recent years numerous projects have addressed one or more of the issues mentioned above. To meet the real-time requirements of embedded systems, for example, advanced scheduling techniques have been developed (e.g., rate monotonic scheduling [37] and best effort decision making [20]). The requirements of distribution, on the other hand, are addressed by new and more powerful networking hardware and communications protocols such as the Open Systems Interconnection Model [33]. Reliability and safety are addressed by advanced software features such as distributed nested transactions [24], while the needs of non-stop operation and dynamic upgradeability [44, 42] are addressed by modular approaches to operating system organization.

Because of the complex way in which the above characteristics are interrelated in embedded systems, however, it is not always possible to use these tools and techniques together in a system which exhibits several, if not all, of these properties. Often a technique which is very successful at solving one particular problem cannot be used with another technique developed to solve another problem because of the way they overlap and interact. The different techniques, and in particular the combination of technologies, have the potential to introduce new problems or exacerbate others. This difficulty is compounded by the fact that systems of this kind are inherently complex and typically very large. In fact, some of the largest software systems to date fit into this category.

For this reason, rather than tackling individual aspects of the problem of supporting the evolution of non-stop, distributed, real-time, MASC systems, the MISSION[1] project has focused on defining the overall development strategy and infrastructure into which such solutions will fit. Specifically, this work has two main thrusts. The first part is to lay the foundation for a new generation of integrated systems software for the target environment in which MASC computing applications are deployed and operated. The second part is to define an accompanying infrastructure which is capable of supporting the construction, verification, reuse and maintenance of the kind of software artifacts required in the target environment. The MISSION approach is believed to be unique in the integration of these advancements across the three environments.

This paper provides an overview of the MISSION approach for supporting distributed, non-stop MASC systems with a particular focus upon the systems software support for applications executing in the target environment. Before describing the approach itself, however, we first describe the main issues that arise in the construction and maintenance of this type of system. In addition to providing a definition and description of each issue, we identify some of the applicable terminology and technologies. The following section then describes the MISSION strategy for dealing with these issues, first introducing the general context in which MASC software is developed, operated and maintained, and then describing the target architecture. We conclude by describing each of the subsystems making up this architecture.

2 Principal Issues

Important issues and requirements for MASC computing systems operating in hostile environments have been discussed in publications such as [1, 14, 36, 38]. This section discusses only five of the principal issues: life cycle approaches; distribution; safety; reliability, security and integrity; and fault tolerance.

Clearly, the requirements for the project as a whole are driven by the target environment. The life cycle requirements for the integration environment, which serves as the site from which the target is monitored, controlled and updated, are principally driven by the need to provide safe and affordable support for the target environment over its complete lifetime. The requirements for both the target and integration environments are, in turn, the principal drivers of the life cycle requirements of the host environment, which is the place where the initial application development and testing takes place. Since the entire set of life cycle requirements for this class of MASC computing applications and systems will probably never be known in advance, an iterative approach to life cycle support is essential.

[1] MISsion and Safety crItical SuppOrt ENvironment

2.1 Life Cycle Approaches

As might be expected, one of the major deficiencies in the current state of the practice for this domain is the lack of predictably-dependable, integrated approaches [11, 23, 29]. Such approaches should be traceable, controllable, and applied iteratively from the system's initial inception through to its retirement. MISSION's goal of defining and verifying such approaches is mirrored in other projects such as Spring [11] and the PDCS project [29].

An important goal of MISSION is to demonstrate that an object-oriented discipline can be used to control the complexity of this MASC target environment. Related issues include the application of the object-oriented discipline to the design of the generic architecture for the target environment systems software. Of particular importance is the evolution of a MASC kernel for this systems software [26, 39, 29]. The kernel is intended to provide a small but powerful set of mechanisms designed especially to support tractable, rigorous reasoning about MASC functions and systems. Support for such reasoning is critical for the infrastructure in the integration and host environments. In addition, safe and affordable approaches should consider the integrated issues of the software, (both applications level and systems level), the hardware, communication links and human-machine subsystems as well as interactions with the environment in which the system is deployed and operated. Techniques currently addressing these system level issues are not well integrated. The Alpha project [26] shares the goal of using the object paradigm to develop systems software that supports tractable, rigorous reasoning about MASC properties.

2.2 Distribution

Providing support for distributed operations is both a problem and an opportunity. Distribution should facilitate new and more powerful forms of fault tolerance along with opportunities to improve performance for real- time command and control systems [30, 41]. Related issues include when and how to assign software components to physical processing sites [5] and what support can and should be provided for migrating components among processing sites [45]. This support must be integrated with the ability to dynamically evolve and reconfigure both the applications and the systems software in the non-stop, distributed target environment (DTE). Unfortunately, no known system currently integrates a full set of acceptable solutions to these requirements with the needed attention to safety.

The need to capture a broad spectrum of information for system objects is even more crucial when real-time decisions are to be made [40]. In a distributed system the universal system state changes faster than can be communicated throughout the system [15]. Furthermore it may never be possible to "snap-shot" a view of the entire system state at any point in time. Decisions therefore must often be made in environments of incomplete and sometimes inaccurate data [20]. The goal of safely supporting dynamic evolution and reconfiguration of non-stop, distributed systems is shared by the Real Time Mach project [43].

2.3 Safety

The following working definition of safety is used in this project "safety is the probability that a system, including all hardware, software, communication links, human-machine subsystems, and interactions with the environment, will provide appropriate protection against the effects of faults, errors, and failures which could endanger life, health, property, or the environment." Safety depends upon related issues such as integrity, reliability, security and others to be discussed in the following subsections. Safety cannot be guaranteed, especially not for the class of MASC computing applications under discussion in this paper. Many important risks, nevertheless, can be managed to improve the probability of sustaining safety across the life cycle [28, 7]. MISSION supports the traditional goal for aerospace applications that no single point of failure can endanger a mission and no two points of failure can endanger safety.

Safety is the most important aspect of any distributed MASC computing system. The system must guard itself against any event or action, intentional or accidental, that compromises its safety [6]. Safety requirements should be considered at each point of the system's life cycle [19, 34].

The ultimate aim of the work reported in this

paper is to define a small but powerful set of constructs that can be used to compose MASC computing applications and systems. These constructs are being defined to support safety properties. Systems composed of such constructs should facilitate tractable, rigorous reasoning about safety. The MISSION project is fairly unique in its emphasis on evolving and verifying approaches to composing safe, non-stop, real time, distributed systems.

2.4 Reliability, Security and Integrity

The safe and affordable support of lives, health, property, environment, and mission in the target environment depend upon system level reliability, security and integrity. System reliability refers to the ability of the system to function under stated conditions for a stated period of time [25], and should be maximized for MASC applications and systems. This requires more than certification of correct software components and highly reliable hardware components. It also requires systems level design for fault tolerance and survivability [16, 31].

System security refers to the protection of the system from accidental or malicious access, use, modification, destruction, or disclosure [9]. Distributed systems which support a diverse group of users are particularly vulnerable to problems which result from improper access to information and other resources. At the minimum, protection is necessary for inadvertent access due to program or operation error. At the other extreme, deliberate disruption must be prevented. The MISSION project seeks to provide security to at least the multilevel security class B3 of the DoD standard for security [9]. Such security should be supported within the target environment and in all its interactions with the integration environment.

System integrity refers to the ability of the system to perform its intended function irrespective of changes in its operational environment [32, 8, 31]. The MISSION approach for ensuring integrity in the target environment builds upon research in executable assertions[35]; monitors [18]; checkpointing and recovery schemes [21]; and distributed, nested transactions [24]. The approach also introduces the concept of the integration environment. These aspects of the approach are discussed in more detail in the following section.

2.5 Fault Tolerance and Recovery

In a perfect world, functionally correct software, hardware, communication links, and human machine subsystems would operate safely and reliably in their intended environment. Unfortunately, in the domain addressed by MISSION, faults, errors, and failures will occur which could be disastrous if not detected and handled properly. MASC systems are needed which can tolerate such problems or, when the problems cannot be tolerated, enact survivability policies.

A failure means that a functional unit can no longer satisfy its requirements at run-time, and may be caused by a defect in the software design or implementation. A fault occurs at run-time and may leave errors in some part of the system, and may sometimes lead to failures. Detection may refer to the detection of either a fault, an error or failure [27]. Recovery refers to the process of restoring normal operation after the occurrence of a fault or failure [21].

Classes of faults, errors, failures, and their combinations should be identified and prioritized according to their probability of occurrence during execution, and the consequences of not properly dealing with them [7, 12]. A safe system is not only able to monitor its status and detect an occurrence of such classes as soon as possible, but can also analyze and control the propagation of the effects and recover safely.

The fundamental issue behind MASC software support is handling the consequences of faults. Two approaches are commonly identified: fault tolerance and fault avoidance. Fault avoidance depends on ultra-reliable hardware, early detection of low-level faults with redundant processing, and the ability to use this redundancy to mask faults in the system from its environment. Specifically, the faults are masked from the system state vectors. Avoidance techniques are valuable but not sufficient [13, 41].

Large, complex systems with intricate dynamic interactions severely limit the ability of fault avoidance to assure safe and correct performance. Even if systems with millions of lines of defect-free code could be built (and they currently can not), they would not execute without faults, errors and failures throughout a long, non-stop, operational

lifetime. Some combination of hardware failure, communication links failure, operator errors, latent software defects or acts of providence will cause problems at runtime. Many of these can be tolerated if the software is built to do so. Others cannot be tolerated but survivability can be maximized if the software is so designed [26].

Fault tolerance is a complementary approach to fault avoidance. Fault tolerance is based upon the assumption that any computation might become defective and result in an erroneous system state vector. Either forward or backward recovery schemes may be used to restore the system to a safe and correct state. Since the possibility for the introduction of such problems exists at all levels of the software hierarchy, it should be considered and addressed at all levels. In so doing, the ability to manage or at least mitigate the effects of faults, errors and failures throughout large and complex systems may be made possible [4, 12, 13, 16, 17, 41]. The MISSION goal to leverage combinations of fault avoidance and fault tolerance in support of MASC requirements is similar to a goal of the MARS project [17].

3 The MISSION Approach

The previous section has described some of the principal issues involved in the construction and maintenance of distributed, non-stop MASC systems. In this section we provide an overview of the MISSION approach for tackling the issues, and integrating the various separate technologies that have been developed to date. In particular, we describe how the MISSION approach addresses the need for precise (semantic) modeling, three computing environments, and a generic architecture for the systems software that executes MASC applications.

3.1 Semantic Modeling

As depicted in Figure 1, the key requirements originate within the distributed target environment (DTE), flow traceably and cumulatively across the integration environment to the host environment and back. System level modeling is fundamental to improved understanding and progress toward safe solutions. The need for such modeling extends beyond the final executing system, and encompasses also the interrelated processes that produce the improved solutions. Such modeling of products and processes has implications for all three environments in the MISSION approach.

A key requirement for an integrated solution is the capability to model system level components and interrelationships among software, hardware, communication links, human-machine subsystems, and their operational environment. The representation of such system level components and their interrelationships should facilitate automated support for tractable, rigorous reasoning about their MASC properties.

To respond to these needs, the MISSION team has adapted an object-oriented modeling approach developed by Embley, Kurtz and Woodfield [10] and augmented the approach with additional semantics in entity- attribute/relationship-attribute (EA/RA) form. The approach by Embley et. al. is based upon a formal definition and depicts object-oriented models in three views. Object-relationship models provide the structural view of the part of the system being modeled. The behavior of each object class that appears in the object-relationship models is depicted in an object-behavior model. Interactions among object classes are depicted in object-interaction models. Although the combination of the three modeling views does support a large degree of tractable, rigorous reasoning about the systems being modeled, the semantics defined in the approach do not provide sufficient granularity to capture all details of interest in the MISSION project. Examples include redundant objects, bindings between software and hardware, workload profiles, reconfiguration of systems resources, etc. An entity- attribute/relationship-attribute (EA/RA) form of representation which has been systematically extended to include object classes, relationship sets, states, transitions, interactions and attributes is a feasible choice for representing these system level components, interrelationships, and their MASC properties. The IRDS standard [3] for this form of semantic representation has been legally extended by the MISSION team to meet these needs. However, a discipline is required to systematically address the inherent complexity within the problem space. The same discipline should also control the associated com-

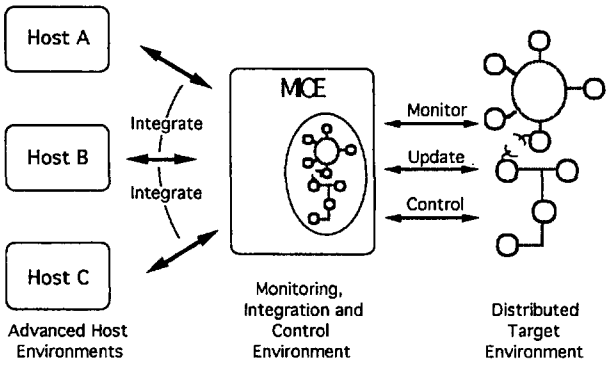

Figure 1: Three Environments

plexity of the processes of evolving and sustaining safe and affordable solutions.

As a scenario to illustrate the modeling discipline and processes advocated by MISSION, consider a proposal to replace and to add types and instances of vehicles in NASA's Space Transportation System. The MISSION process would begin with domain analysis to determine the number of product lines needed (types of vehicles in this example) and the variations needed among instances of each type. Along with attributes such as costs, benefits, risks, opportunities, etc., this "business model" would be captured in object-oriented form and conveyed to the client. Based upon priorities, constraints, and other business and political factors shaping decisions and commitments, the business model would be mapped to a scoping model to identify which product lines and their variations will be evolved, when, and in what order. The object-oriented scoping model would then be mapped to a "concept of operations" model for each product line and its variations. System requirements modeling for the domain would then proceed by revising the concept models to represent common requirements and constraints as-well-as differences among the product lines and their variations. Later, this "domain model" would be mapped to a partitioning and allocation of requirements and constraints among models of: software, hardware and communications, and human interfaces. This stage would be followed by the creation, evaluation and selection of generic architectures appropriate for the domain. The domain engineering process would continue and would eventually be followed by application engineering to create specific instances of the product lines.

Some important points to be noted about this scenario are as follows. First, all products of the process are represented in an extended object- oriented form (i.e., extended via EA/RA notations) whether the products are business models, models of system requirements and constraints, or models of software, hardware and communications, human interfaces, and interactions with the environment. Second, a complete set of semantic information typically requires three views of the object models. Third, tools exist to facilitate such modeling and reasoning about the models. Fourth, the domain engineering processes and the application engineering processes that evolve these products are also represented as object models.

Precise semantic modeling using an object-oriented discipline provides the foundation for constructing system level fault tolerance and avoidance. Systems built from such models can also be designed and verified to enforce policies for survivability when faults and failures occur that cannot be tolerated or avoided. For example, to support fault tolerance, classes of faults, errors, and failures can be identified and modeled for the software, hardware, communication links, human-machine subsystems and operational environment that comprise the intended MASC computing system. Assertions can be formulated to provide context sensitive detection and responses for certain classes of faults, errors, or failures - namely, those classes that are not only likely to occur but which will also produce unacceptable behavior and effects if they not properly handled. One or more monitors to enforce these assertion checks and responses can then be generated to accompany the functional software to the target environment.

Of the research projects that focus on domains overlapping with that of MISSION, MISSION is somewhat unique in its emphasis on process and methodologies that leverage object modeling as a unifying paradigm at the systems level. Alpha shares the commitment to software objects and Spring shares the commitment to tools and methods for the host and the target environment.

3.2 Three Environments

Developers of software for embedded systems have traditionally been concerned with two environments: the host environment (the computers on

which all software requirements analysis, design, implementation, and testing is performed) and the target environment (the embedded computers on which the software is intended to execute). However, these two types of environments are insufficient for MASC systems which are developed by several different organizations, and which are required to execute non-stop. Typically there will be many "host" environments, each used to develop a part of the final system. For example, different host environments could be responsible for different (sub)applications to be added to the existing system. To enable the products from the various "hosts" to be combined, and to provide an interface to the software executing on the target environment, MISSION envisions a third environment - the monitoring, integration and control environment (MICE). The provision of a coherent framework for modeling the structure and behavior of MASC systems impacts all three environments throughout the full life-cycle of the system.

The Monitoring, Integration and Control Environment (MICE), is intended to mitigate the risk in evolving and sustaining remotely distributed, non- stop, MASC computing applications and systems. The MICE serves as an interface between the various hosts and the target environment and is the environment where software from the hosts is integrated. The MICE additionally serves to safely upgrade software components in the target environment, monitor the performance of the target environment, and possibly assist the target environment in performing major reconfigurations in response to faults. To properly perform these tasks the MICE must have up-to-date models of the structure, functionality, behavior and constraints of the elements of the executing target environment. The MICE must also present an appropriate command interface, and provide powerful diagnostic support.

The MISSION project is believed to be unique in its attention to the integration environment within a research context, although environments of this type have historically been an important part of NASA applications (e.g., the Mission Control Center for shuttle operations).

3.3 Generic Architecture

A generic solution architecture is proposed for the domain of MASC computing applications and systems addressed by the MISSION research. As shown in Figure 2, the target environment is a distributed system composed of interacting, multiprocessor clusters. Local area networks (LANs) may be configured from these clusters, and wide area networks (WANs) may be configured from these local area networks. The applications software on each cluster is supported by systems software providing intra- and inter-cluster communication and reliable execution in the presence of component failures. To limit the damage caused by faults, and to increase the feasibility of developing and sustaining such a system, the software on the processor clusters is separated into the following "firewalled" partitions[2] -

1. MASC Kernel
2. Distributed Application System (DAS)
3. Distributed Monitoring system (DMS)
4. Distributed Policy Systems (DPS)
5. Distributed Information System (DIS)
6. Distributed Communication System (DCS)

If, for example, a new space vehicle were required, the number and type of applications and the profile of the intended workload can be used to determine how many clusters (and with what resources), and what LAN and WAN resources will be needed.

Much of the research and development of distributed systems has evolved from an assumption of single processor nodes interconnected by LANs and WANs. Even multiple processor nodes have frequently been configured as "N redundant" processors to avoid certain types of faults. In effect, such processors process a single instruction and data stream with a "voting mechanism" to assure majority rule (e.g., the primary flight control system of NASA's space shuttles).

As a partial result of the "single processor node" mind set, attempts to evolve distributed

[2]By firewalled, we mean that certain steps have been taken to ensure that a fault, failure or error in one partition does not adversely affect other partitions.

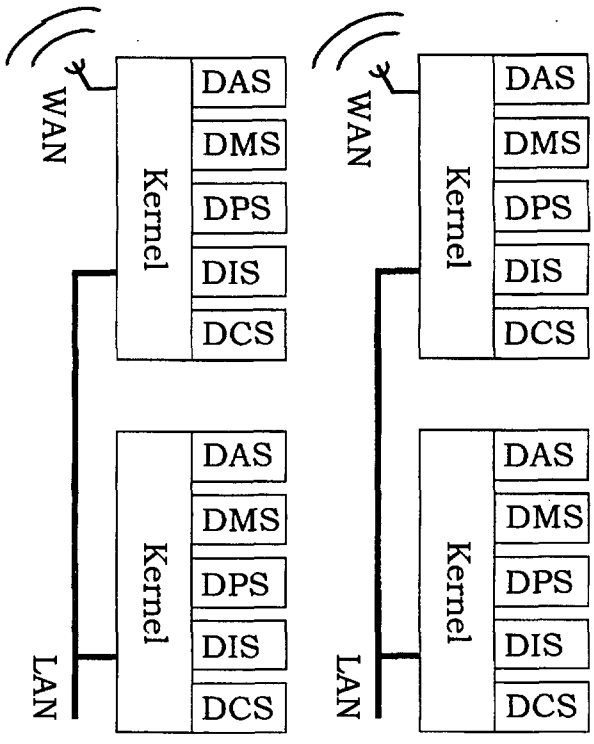

Figure 2: Generic Architecture

systems with tightly constrained, real-time control functions have not been widely successful. Such systems typically experience severe performance problems in meeting their functional requirements. Attempts to integrate a software based approach to supporting systems level fault tolerance tend to exacerbate the overhead problem responsible for the poor performance.

Much of the performance overhead in a single processor node is associated with the time required for context switches. Unfortunately, the elimination of context switches can result in the loss of opportunities to help prevent faults that occur in the execution of one instruction stream from corrupting the subsequent execution of other instruction streams. A key concept of the MISSION approach is to "flatten" the traditional software architecture to take advantage of multiprocessing clusters as illustrated by the cluster architecture in Figure 2. If, for example, such a cluster was located at a geographical site with requirements for four local, hard constrained, real- time control functions, then as many as four or more processors could be assigned to the parallel processing of these control functions. Even if interaction existed among the four functions, parallel processing may offer benefits over a single processor. In the MISSION architecture, the units of functional code are intended to execute in parallel with coroutines on other processors that check for faults, errors, or failures. As long as no flaws are detected, only a minimal performance overhead is added to the execution of the functional code of the applications. Still another performance benefit may be derived by also allowing parallel execution of services and resources that are shared among the applications. For example, persistent information and communications may be organized in such as way as to maximize parallel processing among these subsystems and the applications as indicated in Figure 2.

The MISSION goal to exploit parallel processing capabilities among LANs and WANs of multiprocessing clusters is also a goal of other projects such as Alpha, Spring and Real-Time Mach. The approach to "flattening" the architecture to achieve the intended throughput improvements is particularly evident in Alpha and MISSION.

3.3.1 The Clusters

MISSION clusters have the following properties. Clusters:

- do not share physical memory,
- have access to a hierarchy of memory subsystems including stable storage controlled by transaction mechanisms,
- may be connected to any number of LANs and WANs,
- may have predetermined types of hardware resources, including processors, added to a cluster without changing systems software,
- may fail completely or partially,
- may be repaired and returned to full service, typically without stopping processing,
- may be added/removed at any time,
- may have changes to applications and systems software made without stopping processing, and
- may control access to both physical and virtual systems resources.

3.3.2 The Communications Links

MISSION communications links must be able to tolerate faults, errors and failures which include messages which have lost parts, garbled parts, out-of- order parts, duplicated parts, or parts which are arbitrarily delayed.

3.3.3 MISSION Computing Systems

MISSION computing systems are expected to tolerate, to a specified level, combinations of faults, errors and failures to include: communications failures, abortion of application and system program components, crashes of one or more clusters participating in an application, and lock cycles.

3.4 The Kernel

The MASC kernel is a critical part of the MISSION approach to improving runtime support for the execution and evolution of MASC functions and components in the distributed target environment (DTE). It is similar to the"microkernels" of other projects such as Alpha, Mars and Spring, and provides the foundation on which the firewalled subsystems are built. These mechanisms directly affect the ability of the infrastructure in the integration and host environments to support the DTE. This is because the integrated approaches to semantic modeling are based upon the generic architecture of the DTE systems software.

The kernel is responsible for encapsulating hardware and providing mechanisms to support the policies, operations, and interactions of the other five firewalled partitions. Any communication entering or leaving a partition is a result of invoking the kernel for a message passing service. No direct communication among partitions is allowed apart from with the kernel. The five firewalled partitions, shown in Figure 2, (DAS, DIS, DCS, DPS and DMS) are also referred to as the five firewalled subsystems. Thus, for example a DAS component that wishes to request a resource from the local DIS or from a remote DAS component must invoke a message passing service from the kernel. This modularity allows rigorous reasoning about the kernel independent of the sources or destinations of messages. Since the properties of the structure, functionality, behavior and constraints of the kernel can be assured, the same approach to rigorous reasoning can be independently extended to each of the five firewalled subsystems.

As in [2], MISSION treats the kernel's message-passing relationships with the other subsystems as explicit, first class semantic entities. Protocols are used to describe allowable interactions, their constraints, and their responses to constraint violations, much as in the Mars project. In contrast with the Mars approach, however, MISSION does not assume a clock that is universally available to all clusters in real time.

3.4.1 Twelve Features of the Generic Architecture

The MISSION system architecture embodies twelve features which are either not found at all in today's systems software or are not found as an integrated set. There are at least two important reasons why this set of features is used. First, they facilitate the provision of runtime support needed for the domain of MASC computing applications and systems addressed by the project. Second, they facilitate precise modeling and the associated discipline of rigorous reasoning about the system. These twelve features are identified below:

F1. Model-based reasoning

F2. Firewalled partitions of applications and subsystems

F3. Tailorable interfaces based on classes, objects and messages

F4. Life cycle unique identification of classes, objects and messages at runtime

F5. Extensible and modifiable sets of classes, objects and messages at runtime

F6. Separation of policies and mechanisms

F7. Multiple and adjustable levels of security and integrity

F8. Synchronous and asynchronous communications mechanisms

F9. Adaptable policies for scheduling, redundancy management and the management of other runtime services and resources

F10. Stable storage for checkpointing and recovery

F11. Distributed, nested transactions

F12. "System" level fault tolerance and survivability through systems software.

We elaborate upon these features below.

F1. Model-Based Reasoning

MISSION engineering processes and products emphasize semantically rich, object-oriented models to support tractable, rigorous reasoning about MASC properties. These models can be partially leveraged in the target environment since the kernel contains a finite set of mechanisms designed especially to support the interpretation, maintenance and modification of runtime models. For example, runtime policies are maintained in the DPS as models. In addition, current configuration details are also maintained as on-line models. When an overload condition arises at a cluster, interpretation of the overload policy in terms of the current configuration will determine the response (e.g., load sharing with another cluster or local load shedding).

Although model based reasoning is certainly not new, MISSION is believed to be one of the first projects to investigate its application to nonstop, distributed, MASC systems. Initial studies have focused upon its use in configuration management For example, a resource might initiate one particular recovery response under one set of conditions, and a different recover response under different conditions. Since most elements of the workload and system configuration are well-defined in the DTE models, context sensitive contingency determinations can often be made in parallel with workload processing and be available for rapid response in the presence of one or more anomalies of a predetermined type.

F2. Firewalled Partitions

Firewalled partitions are used in MISSION to maximize the opportunities for identification, isolation, and selection of recovery capabilities. In the host environment, objects are created and assigned to one-and-only-one of the five firewalled subsystems or to the kernel. As the semantic models of the DTE applications and system are evolved, these objects are further allocated to specific clusters. This partitioning and allocation information is exported to the DTE for use by the kernel and the five subsystems. This means that if, for example, an application object executing in a cluster's DAS requests information from an object in the local DIS, the message is passed from the first subsystem to the second by invoking the kernel. Similarly, if an object in the cluster's DAS requests information from a DAS object in a remote cluster, the kernel recognizes that a local object is requesting information from a remote object and invokes the appropriate operation. The message is passed to the local DCS where a communication object will prepare to effect the remote communication.

The result of this organization is to isolate each partition of objects by explicit message passing through the kernel services. For example, suppose a DAS object passes a message to a DIS object which accepts the message and then fails. The opportunities for tolerating the failure are enhanced since the DAS object was preserved in a healthy state when the message was sent. In much the same way, different applications within the DAS, different information systems within the DIS, etc. are also protected from corruption within their own subsystems.

F3. Tailorable and Extensible Interfaces

Dynamic extensibility and other forms of dynamic reconfiguration are facilitated by this feature. Each segment of the generic architecture for the DTE systems software interacts with other segments of the local cluster and with peers in remote clusters through carefully defined interfaces. These interfaces are specified in CIFOs (Catalogues of Interface Features and Options). The interfaces are tailorable in that the given set of applications and system requirements for a given cluster determine which features and options will be selected as CIFO subsets for each cluster. The interfaces are extensible in that precisely modeled rules exist for extending these CIFOs as needed over time. As an example of such rules, no device driver can be replaced until certain preconditions are satisfied such as: "Complete all input/output

operations in progress when the replacement command arrives until a 'recoverable' state is reached. Then effect the replacement."

F4. Life Cycle Unique Identification

This feature also supports tractable and rigorous reasoning about the MISSION models. In the DTE, classes templates, executable images of objects and messages are uniquely identifiable. For example, suppose an object is a part of an application that requires about five minutes to complete and that is intended to run every hour on the hour. The executable image retains its unique identification but, in addition, each hourly activation receives a different thread-of-control identifier. Each thread assignment is provided a unique identification so that the effects of each activation are traceable. Similarly, an iterative object structure may complete and send the same message structure many times during the life span of each object. In the MISSION approach, the effects of each message are intended to be traceable through the unique identifiers of each message, source, and destination(s). The element of the MISSION approach has been strongly influenced by the work of Moss [24].

F5. Extensible and Modifiable RunTime Sets

This feature complements all the preceding features, but is particularly germane to: "F3. Tailorable and Extensible Interfaces". The ability to tailor and extend CIFOs in the host and integration environment is important, but a corresponding capability is needed for objects inside any cluster partition of an operational, non-stop DTE. More specifically, the interfaces to each segment of a cluster architecture should allow existing class definitions internal to the segment to be modified or new ones to be added. Once the modified or new class definitions are installed, the interfaces should encapsulate the ability to create new objects and messages of the new and modified classes. In addition, the interfaces should support the retirement and replacement of old classes, objects, and messages as needed. This mechanism is analogous to the polymorphism/dynamic binding mechanisms of object-oriented languages

F6. Separation of Policies and Mechanisms

This feature not only facilitates tractable, rigorous reasoning, but it also facilitates the domain and application engineering processes through separation of concerns. The MISSION approach partitions and allocates policies to various members of the firewalled subsystems. The shared mechanisms used to effect these policies are in the kernel. For example, the DPS is intended to contain polices for the management of shared services and resources within and among clusters. These policies are encapsulated within DPS modeling objects. The effects are somewhat analogous to earlier techniques of operating systems enforcing "table driven" policies. The interpretation and enforcement of the policies encapsulated by the firewalled subsystems is dependent upon the utilization of the kernel mechanisms. This feature is also supported in Alpha.

F7. Multilevel Security and Integrity

All threads-of-control are created, assigned, sustained and retired via the MASC kernel. A requirement for each active object (i.e., one with its own thread-of-control) is to maintain a registration of its unique identity and its current capabilities. This is particularly important when the active object is about to request a service of another object. A unique identity is required for the destination object and its services and resources. In addition, two other points should be noted. First, the match of a sender's capabilities to a receiver's list of required access rights should be enforced for each access. Second, these rights may sometimes have to be temporarily sacrificed in the cause of higher level policy issues related to a system's fault tolerance and survivability.

F8. Synchronous and Asynchronous Communications Mechanisms

The domain of interest to MISSION researchers includes applications requiring telemetry data to be broadcast as it becomes available and without regard for the status of intended receivers at the time of the broadcast. The domain also includes applications such as multidimensional collision avoidance and proximity operations that require hard constrained, real time synchronization

and control. The literature on communication mechanisms to support distributed and concurrent processing requirements reveals two distinct solutions with certain advantages claimed for each [22].

The first type of mechanism supports the use of asynchronous transmissions and receptions without blocking the sending process or the receiving process(es). Instead, transmission is a case of "send when ready and then proceed". Reception is a case of "receive when ready, if message is available, and then proceed". Variations of this type of mechanism have also been studied.

The second type of mechanism is used for two distinct cases of synchronous communication. The first case involves an active object which calls for a service from a passive object (a passive object borrows its thread-of-control). This case is analogous to a local thread-of-control in a "main" procedure calling a remote subroutine. That is, the thread and its request are passed to the environment of the called subroutine. After "borrowing" the thread-of- control to execute, the passive object returns both the results and the thread- of-control to the calling environment.

The second case of synchronous communications involves a need for synchronization and exchange of information among two-or-more cooperating, active objects. This case addresses, among other things, the issues of the Ada rendezvous among two cooperating threads-of-control. This support for multiple forms of communications is very different than the approaches taken in many other related projects such as Mars which only use datagrams.

F9. Adaptable Runtime Services and Resources

The provision of shared system services and resources to an evolving collection of applications is intended to be based upon well-defined policies, configurations and circumstances. Some resources and services will be replicated to maximize availability and fault tolerance. Such redundancy will need to be managed at a variety of levels. At one extreme, the redundant copies could be managed as "hot standbys" which are ready to be substituted for the primary copy at any time. At another extreme, the redundant copy can be substituted for the primary copy only after processing is performed to prepare the "cold standby" to take over. Depending upon criticality, workload, and the status of system resources, the type and amount of redundancy is intended to vary according to adaptable policies.

Another important aspect of adaptable policies is scheduling. Some real-time applications map naturally to a collection of periodic processes. Others are interrupt driven and are aperiodic. Still others have sporadic service requirements that may be of varying frequency and duration. An important aspect of the approach, therefore, is the use of adaptable scheduling policies to maximize support for MASC functions and components under conditions that vary from normal to various types of emergencies. A similar feature is also found in Real Time Mach.

F10. Stable Storage

Fault tolerance among clusters of distributed MASC systems benefits from the next feature, distributed nested transactions. However, implementation approaches to such transactions require stable storage. Stable storage has two characteristics that facilitate check pointing and recovery. First, it survives temporary losses of power. Second, it is always updated in an atomic operation.

F11. Distributed, Nested Transactions

Fault tolerance among interactive, distributed processing clusters is facilitated by support for distributed, nested transactions [24, 26]. This is particularly true when a fault, failure or error can not be detected in a single state vector, but depends instead upon detection of incorrect sequences of processing. Transactions bracket a named collection of operations between "Begin transaction X" and "End transaction X". The effects of the transaction are to make the set of enclosed operations appear to be a single atomic action. That is, either all of the operations complete successfully or the system can detect and recover from the effects of partial completion. Distributed transactions support hierarchies of parallel and distributed operations. Nesting allows higher level transactions to be composed of sets of enclosed transactions. Transactions of this kind can be used to provide fault tolerance and survivability

in the DTE, and also facilitate reasoning in the host and integration environments. Other related projects employing this mechanism include Alpha and Mars.

F12. System Level Fault Tolerance and Survivability

The MISSION approach leverages systems software to support true systems level fault tolerance and survivability. Since object classes and relationship sets are used to model software, hardware and communications, human interfaces, and interactions with the environment, systems software monitors can be used to monitor and control systems level resources as appropriate.

An important component of the MISSION approach is the concept of coroutines which associate monitors in the DPS with functional objects elsewhere in the system. The job of the monitors is to detect faults, errors, and failures as soon as possible and to then provide support for effecting isolation, analysis, and recovery. Such detection is based upon assertions that are associated with MASC properties. These assertions may be about values of state or about sequences of state transformations. The Mars project also employs kernel-level mechanisms to support system-level fault tolerance.

3.5 Firewalled partitions

As mentioned above, and illustrated in Figure 2, the generic architecture employs five firewalled partitions that interact by means of the message passing services provided by the kernel. In this subsection we outline further the role of each subsytem.

Distributed Applications System

The DAS is the firewalled subsystem containing MASC applications that are to be executed on the MASC computing system. The focus of the research in the DTE is on the generic architecture of the systems software rather than the DAS. The DAS developers are intended to leverage the features and options of this generic architecture to improve runtime support of MASC functions and components.

Only two aspects of the DAS are within the scope of this research project. The first is the set of interfaces to the local cluster and to DAS peers in remote clusters. The second is the set of abstractions made available to applications programming teams to improve safety and affordability. However, another important point should also be understood about a DAS partition of a cluster. Any component within a DAS application is firewalled from the other partitions and from other applications within the DAS. That is, different applications and partitions have no direct means of communication, but must invoke a message service of the kernel. This additional firewalling of applications is also supported within the other partitions and is used to facilitate tractable, rigorous reasoning about the individual parts of a partition.

Distributed Information System

The DIS is responsible for managing shared and persistent information services and resources. Whenever information is shared by more than one application, access to the information is provided via a virtual interface set by requesting services from the DIS. For example, a DAS component could request a unit of shared information from the DIS by invoking a message service from the kernel. Also, some applications do not execute continuously and have requirements for persistent information. For example, a program that takes five minutes to complete may be scheduled to execute once every eight hours. At each execution, the program updates some information in the DIS that must persist between executions. In addition, the DIS manages shared and persistent information on behalf of the systems software. Examples include: performance and workload by cluster, LAN, WAN, and system; health and status of ..., etc. As with the other firewalled partitions, portions of multiple DISs may reside on the same cluster. Each DIS represented on the cluster is firewalled from the other DISs also on the cluster.

The class of MASC computing applications and systems addressed by MISSION will typically be long lived. Many type definitions that will be needed in the future cannot be known when the system is initially developed and deployed. Since non-stop operation requirements prohibit brin-

ging the system down to recompile existing code in the context of the new type definitions, an alternative is needed to upgrade the system. The approach under study is based upon controlled inheritance. A set of commands in the Distributed Command Interpreter is intended to allow the MICE to first extend/add the definitions and then create instances of the types. The reader should note that the problem of dynamic type extensibility is not limited to just the DIS.

Distributed Communications System

The DCS corresponds to the upper three layers and a portion of the fourth layer of the seven layer ISO model for Open Systems Interconnection [33]. (The lower layers are encapsulated as device drivers within the kernel.) The DCS is responsible for managing communications services and resources among clusters, LANs and WANs. Within a cluster, whenever an applications component or a systems software component needs to communicate with a peer at another cluster, the DCS is responsible for effecting this communication. A virtual interface set shared with its DCS peers at other clusters is used to resolve issues of routing, congestion control, relocation, and other services. Such resolution is transparent to the applications components or to any systems software components located outside the DCS partition.

Distributed Policy System

The DPS is responsible for the evolution and enforcement of policies regarding the sharable services and resources of the integrated systems software. The DPS contains a library of policies which are used in conjunction with the mechanisms of the kernel to manage such issues as: contention between local cluster priorities and universal system priorities, multiparameter scheduling, emergency load shedding, dynamic reconfiguration and others. An important premise is that support can be predictably and dependably provided for different policies needed by different applications if a known set of sufficient resources are available and if a known set of universal and local policies permit. This is somewhat similar to the approaches taken in Alpha and Spring.

Distributed Monitoring System

One of the most important and unique features of the MISSION "smokestack" model is the distributed monitoring system. This contains the objects responsible for monitoring the correct execution of the application objects. In fact, monitor objects are also introduced to monitor the correct execution of system level objects.

For any MASC component or any set of communicating MASC components in any of the other firewalled partitions, the engineers in the host environment are responsible for identifying those classes of faults that must be tolerated or that must invoke survival policies. Context sensitive assertion checks can then be generated to detect such faults at run time, and handlers can be prepared to respond to such detections. These assertions and handlers can then be combined into monitors. Together with policies in the DPS, they are responsible for system level fault tolerance and survivability.

When a work module (i.e. an application or system module) is installed in the DAS, or other appropriate partition, the corresponding monitors are installed in the DMS. The work module and associated monitors are scheduled to run concurrently on separate processors, although the work module is modified to write key information about state values and state changes to designated bulletin boards as it executes. The monitor is programmed to read this information for its assertion checks, and as long as no violations are detected, the work module is allowed to continue. However, if a violation is detected, the corresponding policy is consulted and the appropriate handler is invoked. If the fault is entirely local to a single work object, then the associated monitor may be able to insure proper tolerance by itself. However, faults that will cause temporal, spatial, or value errors in other objects or faults among cooperating objects are addressed by monitors that coordinate the activities of the monitors of the affected objects (i.e., monitors that monitor and coordinate other monitors).

Another primary function of the DMS is to be the "window" to the target environment for the MICE. Under normal operation, the DMS will monitor the health and status of the clusters, LANs and WANs and report this information (via the DCS) to the MICE. Other normal facilities

that it will control or monitor include: the introduction or removal of a new object to a cluster; the movement of an object from one cluster to another; linking, loading and starting new programs downloaded from the MICE; suspending or aborting threads of control; etc. Although other projects such as Mars have explored the use of monitors for fault detection at the cluster level, MISSION is somewhat unique in its use of monitors to detect faults and coordinate recovery among multiple applications spanning multiple clusters.

4 The Testbed

The MISSION testbed uses Sun workstations for the host and integration environments. A version of a Verdix Ada (1983) compiler that supports post- partitioning and distribution of code has been used to generate code for the target environment. Although other processor types have been successfully used in this target environment (e.g., object-oriented processors by Ericsson), the major clusters consist of multiprocessing clusters of Motorola 68030s.

5 Conclusion

Distributed, non-stop MASC systems are some of the largest and most complex computer systems to have been tackled to date. As described in section 2, they require many independent technologies, developed separately for smaller systems, to be brought together and integrated into a single unified whole. This integration, and the definition of the environment to support it, presents a major technological challenge. This paper has outlined some of the major issues which arise in the construction and maintenance of this category of embedded system, and has provided an outline of the MISSION approach to achieving this goal. This strategy has two principal components: the definition of a generic architecture for the target systems software, and the design of a supporting infrastructure and processes.

A key component of the proposed infrastructure is the monitoring, control and integration environment (MICE) which bridges the gap between the traditional host and target environment used today for embedded systems. The MICE serves as the location at which new software components and (sub)applications from the various contractors can be tested, assembled and eventually downloaded to become part of the executing MASC embedded system. To perform this function the MICE employs a set of precise semantic models which describe the current structure, functionality and behavior of the executing system. Such semantic modeling pervades all three environments, over the full life-of the system, and forms the cornerstone of the MISSION software process used to develop and sustain distributed, nonstop, MASC systems. Each process is domain-specific and leverages the object paradigms for modeling all aspects of the systems across the life cycle.

The paper also outlined the nature of the generic architecture for the multi- processor clusters, interconnected by LANs and WANs, which make up the distributed target environment. This architecture is based on the principal of segregating functionally cohesive components into separate, firewalled, partitions which can only interact indirectly via the special MASC kernel. Preliminary prototypes of these subsystems have demonstrated the feasibility of the architecture and the overall approach, but further work is needed to elaborate upon the detailed make up of the separate subsystems, and to evaluate the concepts in a pilot project.

Acknowledgments

The MISSION research has been partially supported by NASA. The authors wish to thank: the NASA sponsors and monitors; our fellow researchers from the faculty, staff and students at the University; our fellow researchers from industry; our support staff in RICIS; and the volunteers on our Industrial Advisory Committee.

References

[1] AIA/SEI (Aerospace Industries Association/Software Engineering Institute), *Workshop on Research Advances Required for Real Time Software Systems in the 1990s*, Software Engineering Institute, 1991.

[2] Allen, R., D. Garlan, "Formalizing Architectural Connection", *Proceedings of the 16th In-*

ternational Conference on Software Engineering, Sorrento, Italy, 1994.

[3] "American National Standard Information Resource Dictionary System", American National Standards Institute, Group X3H4, New York, 1985.

[4] Arlat, J., K. Kanoun, J. Laprie, "Dependability Modeling and Evaluation of Software Fault Tolerance: Recovery Blocks, N Version Programming, N Self Checking Programming", *First Year Report on Predictably Dependable Computing Systems*, Volume 3 of 3, Esprit Project 3092, 1990.

[5] Atkinson, C., T. Moreton, A. Natali, *Ada for Distributed Systems*, Ada Companion Series, Cambridge University Press, 1988.

[6] Burns, A. and C. McKay, "A Portable Common Execution Environment for Ada", *Ada: The Design Choice - Proceedings of the Ada- Europe International Conference*, Madrid, 1989, Cambridge University Press, 1989.

[7] Charette, R., *Software Engineering Risk Analysis and Management*, McGraw Hill, 1989.

[8] Deswarte, Y., J. Fabre, J. Laprie, D. Powell, "A Saturation Network to Tolerate Faults and Intrusion", *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, IEEE Computer Systems Press, 1986.

[9] DOD (Department of Defense, United States of America), "Trusted Computer System Evaluation Criteria", *DOD 5200.28-STD*, 1985.

[10] Embley, D., B. Kurtz, S. Woodfield, *Object-Oriented Systems Analysis: A Model-Driven Approach*, Yourdon Press, 1992.

[11] ESPRIT (European Strategic Program for Research and Development in Information Technology), *First Year Report on Predictably Dependable Computing Systems*, Volumes 1, 2, 3, ESPRIT, 1990.

[12] Ezhilchelvan, P. and S. Shrivastava, "Characterization of Faults in Systems", *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, IEEE, 1986.

[13] Ezhilchelvan, P. and S. Shrivastava, "A Distributed Systems Architecture Supporting High Availability and Reliability", *Proceedings of the 2nd International Working Conference on Dependable Computing For Critical Applications*, IEEE, February 1991.

[14] GAO (Government Accounting Office), "Space Station: NASA's Software Development Approach Increases Safety and Cost Risks – Report to the Chairman", Committee on Science, Space and Technology, House of Representatives, GAO, 1992.

[15] Jensen, E., Chapter 8, *Distributed Systems: Architecture and Implementation*, (B. Lampson, M. Paul and H. Siegert, editors), Springer-Verlag, 1981.

[16] Knight, J. and J. Urquhart, "On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems", *IEEE Transactions on Software Engineering*, Vol SE-13, No. 5, May 1987.

[17] Kopetz, H., A. Damm, C. Koza, M. Mulazzani, W. Schwabi, C. Senft, R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach," IEEE Micro, February 1989.

[18] LeBlanc, R. and A. Robbins, "Event Driven Monitoring of Distributed Programs", *Proceedings the 5th International Conference on Distributed Computing Systems*, IEEE 1985.

[19] Leveson, N., "Building Safe Software", *Proceedings of COMPASS*. 1986, IEEE, 1986.

[20] Locke, D., "Best Effort Decision Making for Realtime Scheduling", *CMU-CS-86-134*, Carnegie Mellon University, 1986.

[21] Long, J., W. Fuchs, J. Abraham, "Implementing Forward Recovery Using Checkpointing in Distributed Systems", *Proceedings of the 2nd International Working Conference on Dependable Computing For Critical Applications*, IEEE, February 1991.

[22] McKay, C. W. and C. Atkinson, *Volumes I, II and III of the MISSION Concept Document*, RICIS Report, University of Houston-Clear Lake, 1992.

[23] McKay, C., D. Auty, K. Rogers, "A Study of System Interface Sets (SIS) For the Host, Target, and Integration Environments of the Space Station Program (SSP)", SERC(UHCL) Report SE. 10, NCC9-16, 1987.

[24] Moss, J., *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT/LCS/TR 260, Massachusetts Institute of Technology, April 1981.

[25] Musa, J., A. Iannino, K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw Hill, 1987.

[26] Northcutt, J., *Mechanisms for Reliable Distributed Real- Time Operating Systems: The Alpha Kernel*, Academic Press, Boston, 1987.

[27] Parker, D., G. Popek, A. Rudison, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, C. Kline, "Detection of Mutual Inconsistency in Distributed Systems", *IEEE Transactions on Software Engineering* Vol. SE-9, No. 3, IEEE, May 1983.

[28] Pyle, I., *Developing Safety Systems: A Guide Using Ada*, Prentice Hall, 1991.

[29] Ramamritham, K., J. Stankovic, *Overview of the Spring Project*, University of Massachusetts Amherst, COINS Technical Report 89-03, January 1989.

[30] Randall, C., P. Rogers, C. McKay, "Distributed Ada: Extending the Runtime Environment for the Space Station Program", *Sixth National Conference on Ada Technology*, March, 1988.

[31] Randell, B. and J. Dobson, "Reliability and Security Issues in Distributed Computing Systems", *Proceedings of the 5th Symposium on Reliability in Distribute Software and Database Systems*, IEEE, 1986.

[32] Redmill, E. (Editor), *Dependability of Critical Computer Systems 2*, Elsevier Applied Science, 1989.

[33] "Reference Model of Open Systems Interconnection". *ISO/TC97/SC16/N227*, International Standards Organization, 1979.

[34] Rogers, K., M. Bishop, C. McKay, "An Overview of the Clear Lake Life Cycle Model (CLLCM)", *Proceedings of the 9th Annual Conference on Ada Technology*, ACM, March, 1991.

[35] Sankar, S., "Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs", *STAN-CS-89-1282*, Stanford University, 1989.

[36] SATWG (Space Avionics Technology Working Group), Space Avionics Requirements Study, (Integrated by General Dynamics), 1990.

[37] Sha, L. and J. Goodenough, "Realtime Scheduling Theory and Ada", *IEEE Computer*, April 1990.

[38] Shankar, K., C. McKay, "Why NASA, Code R, Should Sponsor Advanced Research in Software Engineering: A White Paper", *Proceedings of the Computing in Aerospace 9 Conference, American Institute of Aeronautics and Astronautics*, San Diego, October 1992.

[39] Stankovic, J. and K. Ramamritham, "The Design of the Spring Kernel", *Proceedings of the Real Time Systems Symposium*, IEEE, December 1987.

[40] Stankovic, J., K. Ramamritham (editors), Tutorial: Hard Real Time Systems, *IEEE Computer Society Press*, 1988.

[41] Strigini, L., "Software Fault Tolerance", *First Year Report on Predictably Dependable Computing Systems*, Volume 2 of 3, Esprit Project 3092, 1990.

[42] Tindell, K., "Dynamic Code Replacement and Ada", *Ada Letters*, Vol. X, No. 7, 1990.

[43] Tokuda, H., T. Nakajima, P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System", *Proceedings of the Usenix Machine Workshop*, October 1990.

[44] Vincente, B., A. Alonso, J. Amador, "Dynamic Software Replacement Model and It's Ada Implementation", *Proceedings of TriAda'91*, ACM, 1991.

[45] Zicari, R., "Operating System Support For Software Migration in a Distributed System", *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, IEEE, 1986.