Semi-Annual Progress Report
Award No. NAG-1-260
March 5, 1982 - February 14, 1988

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Mr. C. Michael Holloway
ISD M/S 125

Submitted by:

J. C. Knight
Associate Professor

Report No. UVA/528213/CS88/111
August 1987

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

Semi-Annual Progress Report
Award No. NAG-1-206
March 5, 1982 - February 14, 1988

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Mr. C. Michael Holloway
ISD M/S 125

Submitted by:

J. C. Knight
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERNG AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528213/CS88/111
August 1987

Copy No. 4

CONTENTS

SECTION 1

INTRODUCTION

The primary goal of this grant is to investigate the use and implementation of Ada* in distributed environments in which reliability is the primary concern. In particular, we are concerned with the possibility that a distributed system may be programmed entirely in Ada so that the individual tasks of the system are unconcerned with which processors they are executing on, and that failures may occur in the software or underlying hardware. A secondary interest is in the performance of Ada systems and how that performance can be gauged reliably.

Over the next decade, it is expected that many aerospace systems will use Ada as the primary implementation language. This is a logical choice because the language has been designed for embedded systems. Also, Ada has received such great care in its design and implementation that it is unlikely that there will be any practical alternative in selecting a programming language for embedded software.

The reduced cost of computer hardware and the expected advantages of distributed processing (for example, increased reliability through redundancy and greater flexibility) indicate that many aerospace computer systems will be distributed. The use of Ada and distributed systems seems like a good combination for advanced aerospace embedded systems.

A distributed system is only as reliable as its weakest node. For example, if each of three computers in a system will operate correctly 99 percent of the time, the complete system will function, on average, only approximately 97 percent of the time. If a node has failed, however, a distributed system has the potential to continue providing service since some hardware facilities remain. Continuation will make these systems *more* reliable than single-processor architectures.

* Ada is a trademark of the U.S. Department of Defense

In order to be truly useful from a reliability standpoint, however, distributed systems must be safe: they should provide a level of service after failure which reflects the reduction in computing power and does not jeopardize the fundamental purpose of the application. Post-failure operation could take the form either of ongoing but reduced service (as in an aircraft control system) or of a timely, controlled shutdown (as in a nuclear power plant).

Such real-time control applications as are found in the defense and aerospace industries often require very high reliability, so toleration of hardware failures can be extremely important. We designate such applications to be *crucial* in that their failures may cause human lives to be endangered or enormous sums of money to be wasted.

A concern that is emerging, however, is the performance that might be achieved from Ada implementations. Initial performance measures indicate that there is considerable variability between implementations and between different types of program on the same implementation. Programmers have expressed concern that, although Ada may allow software to be developed with fewer faults, the execution-time performance may make the use of Ada impractical.

In previous work under this grant we have shown the general inadequacy of Ada for programming systems that must survive processor loss. We have also proposed a solution to the problem in which there are no syntactic changes to Ada. We felt confident that the solution was adequate but could not be sure until the solution was tested. A major goal of this grant therefore was and is to evaluate the approach using a full-scale, realistic application. The application we are using is the Advanced Transport Operating System (ATOPS), an experimental computer control system developed at NASA Langley for a modified Boeing 737 aircraft. The ATOPS system is a full authority, real-time avionics system providing a large variety of advanced features.

The preliminary evaluation lead us to conclude that the approach was indeed workable. We documented the preliminary results in a paper presented at the 1987 Washington DC Ada symposium. That paper has been supplied to the sponsor under separate cover. Although workable, various deficiencies in style and flexibility were noted and a new approach was devised that is far more useful than the original. A preliminary discussion of that approach is presented in sections 2, 3, and 4 of this report.

In a new effort under this grant we have begun work to provide performance analysis of Ada implementations. Our goal is to supply the system designer with tools that will allow a rational decision to be made about whether a particular implementation can support a given application early in the design cycle. A commitment to the use of Ada is viewed as a substantial risk by most project managers mainly because of unknown performance, and if this risk could be reduced it would enhance the use of Ada considerably. A discussion of this project on performance has been supplied to the sponsor under separate cover.

Thus, during this grant reporting period our primary activities have been:

(1) Analysis of the original approach to recovery in distributed Ada programs using the ATOPS example.

(2) Review and assessment of the original approach which was found to be capable of improvement.

(3) Preparation and presentation of a paper describing this work at the 1987 Washington DC Ada Symposium.

(4) Development of a refined approach to recovery that is presently being applied to the ATOPS example.

(5) Design and development of a performance assessment scheme for Ada programs based on a flexible user-driven benchmarking system.

A list of papers and reports prepared under this grant, other than the annual and semi-annual progress reports, is presented in the Appendix.

SECTION 2

ANALYSIS OF PREVIOUS APPROACH

Ada [1] has been designed for programming crucial, real-time applications and for the development of programs distributed over multiple computers. Unfortunately, we have shown it to be deficient in three areas [2]. First, the Ada Language Reference Manual (LRM) fails to specify adequately the facilities available to a programmer who wishes to control the distribution of the various elements of a program. Second, the LRM lacks any clear definition of the *meaning* of a distributed Ada program. Specification of the basic units of distribution, whether they be individual statements, tasks, packages, or anything else, is absent from the document. Third, the LRM does not define the state of a distributed program after the loss of a portion of its computing environment. The implication is that the Ada machine does not fail.

In short, Ada lacks both distribution and failure semantics. As a result, no way exists of ensuring that one distributed Ada implementation is compatible with another. Worse still, the problem is not simply one of defining a standard set of semantic enhancements: the issues are difficult and controversial as well.

With the notion that the Ada machine does not fail in mind, some researchers have proposed a *transparent* [3] approach to continuation in which an application program is unaware of the fault-tolerant capabilities of the system. Loss of a processing node would cause automatic reconfiguration of the system, and the reconfiguration would be invisible to the program. The burdens of recovery and of the preparation needed for that recovery devolve upon the execution-time environment. Because this approach does not allow the application program to be aware of failure or of reconfiguration, the level of service provided after the failure must be identical to that provided during normal operation. Programs that provide transparent continuation require

extra computing power in order to operate as desired after a failure. Also, because the continuation facilities cannot be tailored to the real needs of the application, the transparent approach incurs substantial overhead.

In a paper prepared under this grant [2], we have suggested a method by which system designers can specify explicitly the service to be offered following node failure. This allows for the specification of degraded service in systems which, due to lack of computing power, cannot provide full functionality after a hardware failure. In addition, with this approach designers control the normal operational overhead required to prepare the software for an arbitrary hardware fault. This overhead consists primarily of transmissions of critical data items required during reconfiguration to bring the system to a consistent state. This method is termed the *non-transparent* approach to tolerating the loss of a processor in a distributed system.

We are convinced that only a non-transparent continuation strategy is appropriate for real-time applications which have stringent performance constraints. On a military aircraft, for example, there may not be enough power to double the number or size of the computers in order to provide hardware fault tolerance. There may not even be sufficient physical space and weight may be limited.

We are now equally convinced that the semantics proposed in the original approach to provide the capacity to construct fault-tolerant programs are insufficient and unnecessarily unwieldy. Our objections to that approach are detailed fully in the next section. Basically, we have settled upon a compromise, defined and supported in the next section, between the programming convenience, inflexibility, and high overhead of transparent continuation and the complexity, flexibility and low overhead of non-transparent continuation.

In the remainder of this section, we define three basic assumptions about the hardware architectures that we consider in the remainder of this report and we also define some basic

qualities of the software architectures with which we are concerned. In the next section, we describe the revised semantic enhancements to Ada which we deem necessary for the construction of fault-tolerant software. In section 4, we describe the criteria according to which we evaluate applications of the suggested semantics. In section 5, we develop a software construction strategy which employs the proposed semantics.

Assumptions

In order to focus on the basics of the fault toleration method at hand, we make three simplifying assumptions. First, for the following three reasons, we assume that the system is fully connected:

(1) **No connective exclusivity.** Fault-tolerant design of systems whose individual nodes have exclusive I/O or internode connections is difficult if not impossible. After failure of a node with exclusive access to the effectors, for example, the remainder of an aircraft control system must proceed without control over the aircraft surfaces.

(2) **Low network costs.** Fiber-optic technology is making fast, inexpensive networks available to everyone. With a crucial system, the additional network cost of full connection should always be much lower in priority than the benefits of fault tolerance.

(3) **Flexibility.** A fully connected system has the property that each node in the system sees the same environment. All processes, including those which communicate with the outside world, can, theoretically, be redistributed with impunity. This inherent lack of distribution constraints allows us to examine more easily the effects of software continuation mechanisms on overall distribution flexibility.

Our second assumption is that the system is a homogeneous collection of processors. This simplifies the problem of fault toleration in two ways:

(1) **No special-purpose processors.** We recognize the need for special purpose processors in certain distributed systems. For example, a system might have only one node which contains floating point hardware. The designers have to make allowances in their recovery plan for the extra lost processing power if the "special" node fails. This is not a trivial consequence of using special-purpose processors, because we will show later that providing the same level of service after any node failure simplifies the problem significantly. We feel justified in ignoring the issue for two reasons. First, the problem is not specific to our brand of fault tolerance; a special-purpose machine should be replicated in any system which cannot survive its loss. Second, a crucial system, one for which failure is unacceptable, should always be funded well enough to provide such replication.

(2) **No portability problems.** One of the major concerns with distributed software is flexibility. Ideally, addition of a processing node in response to increased computing needs should require of the software no more than alteration of the distribution directives. Heterogeneous systems, those whose nodes are not identical, must cope not only with redistribution but also with portability. Because of compiler-specific pragmas, we do not anticipate portability of Ada programs to be perfect; however, the requirement that Ada compilers be validated in order to be licensed should cause portability costs to be low. We surmise that software portability will not be affected unduly by the fault toleration requirement because we expect portability to be a function mostly of compiler quality and consistency. At any rate, we are not equipped currently to measure portability, so we will avoid the issue for the time being.

In sum, we are concerned with the ideal distributed targets. These are systems which have homogeneous communication and processing structures. We realize that many crucial, real-time systems will operate on more idiosyncratic targets; however, we believe that these simplifications allow us to focus on issues we consider more fundamental. Additionally, we have argued in this section that the crucial aspects of the systems for which fault toleration is a necessity make it reasonable to expect that the benefits of target homogeneity be purchased. Figure 1 depicts this general target architecture in the context of our example application.

Our third assumption is that most systems will only need to tolerate a single, arbitrary node failure. We expect that the hardware for these crucial systems will be extremely reliable, and we believe that in many cases the probability of simultaneous failure will be acceptably low. A key exception is physical damage. If a bomb explodes on an aircraft, the probability of damage to multiple nodes is much higher than could be predicted by the mean times to failure of the

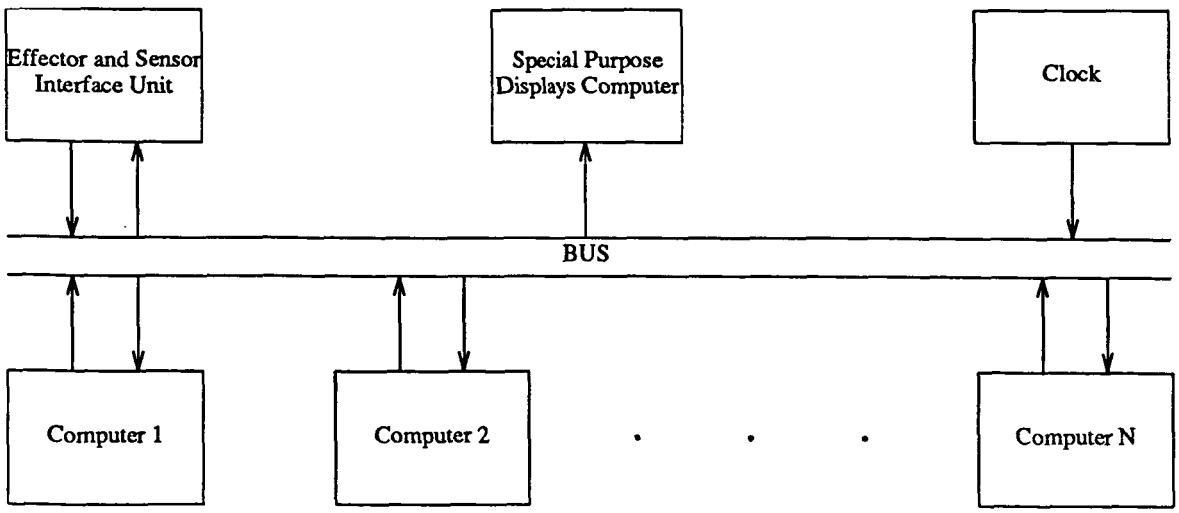

Figure 1 - Distributed Architecture

computers being used. In systems which are deemed susceptible to physical damage, the software requirements may reasonably dictate that some service must be provided even if only one computer remains after a damage incident. By no means do we rule out the use of the semantics defined in the next section for systems that must tolerate multiple failures; however, because of limited resources and in the interests of brevity and simplicity, we do not examine such cases in this research.

Intended Class of Applications

What follows is not so much a definition of *real time* as it is a description of what we mean by the term in this research. Typically, in real-time applications, required functions fall into one of two classes. Some functions need to be executed on a particular real-time schedule, and the remainder may be executed in background. These functions, whether scheduled or background, usually operate in loops which terminate only when the entire system is shut down. Each scheduled function will be associated with a particular frame rate, and each scheduled function will execute once during each of its frames. The background functions will iterate whenever no scheduled functions are executing.

Coordination of the activities of these various functions in time is the problem which distinguishes real-time applications from other programming endeavors. Most programs schedule activities from within. Input, computation, and output occur when the program itself decides, and repeated executions of the program with the same input set will yield identical output sets. Real-time programs, on the other hand, schedule according to external events. An external event can be either a clock interrupt or a burst of input which occurs according to some regular or irregular schedule. In either case, real-time programs must be prepared to respond immediately to new information from the outside world, and, because the timing of events always varies, no two executions of a real-time program will yield the same output set except in the most

improbable of coincidences.

SECTION 3

PREFERRED SEMANTIC ENHANCEMENTS

In a previous paper [2], Ada was shown to lack both distribution and failure semantics. The LRM does not define facilities to effect the distribution of a program on a multicomputer, boundaries on which such a partitioning may occur, or the state of a program after the loss of a portion of its computing environment. Appropriate semantic enhancements to the language were proposed, and a simple strategy for the construction of fault-tolerant systems was outlined. From a software engineering perspective, that strategy lacks both expressive power and grace. Further, those limitations result from the proposed semantics. In this section, we suggest new semantic enhancements that allow for the construction of fault-tolerant programs that meet modern software engineering standards.

Distribution Semantics

In the previously defined approach to continuation, programs must be partitioned along task boundaries. A distributed program would consist of an empty main program and a collection of tasks. Cornhill [3] details an important objection to this strategy:

> Unless we discover that there is something intrinsic to software for distributed systems which causes tasks to exist precisely where we need them for partitioning, we burden the developer with rules that go beyond those specified in the [LRM]. The effect of these additional rules is to define an Ada subset, since only certain legal Ada programs, those that conform to the additional rules, can be executed on a distributed system.

The original reason for the limitation to partitioning only on task boundaries was that Ada tasking semantics can be more easily adapted to fault tolerance than can other portions of the language. In principle, however, as far as the distribution semantics are concerned, we side with Cornhill. Our preliminary attempts to implement the ATOPS application showed the semantics defined in

the previous approach to continuation lead to particularly unwieldy structures.

One structural problem we came across results from the fact that packages, not tasks, are the Ada constructs that are responsible for defining the complex interfaces required for well-engineered solutions to real problems. Consequently, the requirement that tasks be the only partitionable units severely restricts proper application of modern software engineering techniques. Also, since global data is disallowed by the decision to make tasks the only distributable units and because the entries of dependent tasks are not easily made visible in the scope of the parent task, it can be quite difficult to implement shared data between tasks.

We found that the only logistically workable program structure that employs the original distribution semantics is one which mimics a multi-program approach. Tasks would be distributed one to a processor, and they would be responsible for handling all interprocessor and I/O communications. Within each encapsulating task would be a self-contained "typical" Ada program structure which uses packages freely to improve modularity and so forth.

The basic problem with this strategy is that the addition or deletion of a processor necessitates massive program revision, since the program structure is effectively tied to a particular target architecture. This inflexibility is especially serious for fault-tolerant programs, since they must be designed to reconfigure themselves *dynamically* in response to arbitrarily occurring changes in the hardware configuration.

A second structural problem results from the fact that no provision is made for use of software redundancy to reflect the underlying hardware redundancy. Although we do not advocate exclusive reliance on one-to-many mappings to support fault tolerance, we do recognize their utility in some contexts. For example, the cache of critical data items which must be maintained in order to allow recovery from arbitrary processor failure should be implemented as one data structure that is distributed to multiple memories. The reason for this is that this

structure would only be written for the purpose of saving data for use after failure. The overhead associated with the redundancy is therefore not extraneous or wasteful, since each new critical value must be broadcast to the other machines anyway. The big advantage to programming the critical data cache in this fashion is simplicity. In the approach, special tasks on each machine maintain their own uniquely named copies of the cache. What could be a single assignment operation if software redundancy were supported must instead be programmed as a series of rendezvous and assignments.

A generally better approach to the partitioning of Ada programs is discussed in Cornhill [3]. Briefly, the strategy allows for the distribution of most named constructs. The category includes tasks, packages, subprograms, named blocks, and objects (including both indexed and selected components). Neither accept statements nor named blocks that contain accept statements may be distributed†.

The partitioning is expressed in a separate notation which is called the Ada Program Partitioning Language (APPL). The Ada program and the distribution specification constitute separate inputs to a distributed Ada compiler. The notation can express static, dynamic, and one-to-many mappings of distributable program constructs onto the underlying architecture. The one-to-many mapping feature is intended to support fault tolerance. Together, copies behave as if there were only one instance of the given construct. The run-time support system ensures that the illusion is maintained correctly by invoking an inter-copy communication protocol whenever one of the copies communicates with another portion of the program.

We like the general philosophy of APPL. Because an APPL specification remains separate from the Ada program to which it pertains, a distributed program can be developed and tested without considering the details of the multicomputer which will execute it. We note that this

† The assumption is that the internal synchronization of a task should be the province of exactly one processor.

approach has the disadvantage that the partitioning is separate yet the algorithm really needs information about the partitioning if it is to effect reduced service following failure. However, as long as full advantage is taken of the parallelism that is intrinsic to the algorithm, the resulting program can be executed with few changes on a wide range of distributed architectures. In fact, there are two circumstances in which changes might be necessary.

Because APPL is a language in its own right, program-to-processor mappings which would be difficult to define using Ada pragmas and address clauses can be expressed clearly and easily.

Because APPL takes an especially liberal view of what it means for an Ada program to be distributed, the expressive power of Ada is not limited by the multicomputer context. Excepting both accept statements and named blocks which contain accept statements, any named construct can be distributed separately.

This notion of the liberality of APPL deserves some comment. In this research we focus on a tiny (but important) subset of the entire class of distributed applications for which Ada was designed. We must resist the temptation to define distribution and failure semantics which cater only to the concerns of people who develop crucial, real-time, fault-tolerant systems. Any general enhancements to Ada must be generally useful. Ada distribution semantics must be appropriate tools for the development of distributed Ada programs which need not tolerate node failure or run in real time as well as those crucial, real-time applications with which we are concerned. Similarly, Ada failure semantics should be useful for *all* programs which must tolerate partial hardware failure and not just those programs which face stringent real-time constraints. With a flexible, general form of Ada, constraints which are peculiar to a specific kind of application can be introduced at the design level.

Figure 2 depicts this idea. Within the dotted box are the areas of particular concern to us in this research. Outside of the dotted box is a list of at least two (hypothetical) strategies for the

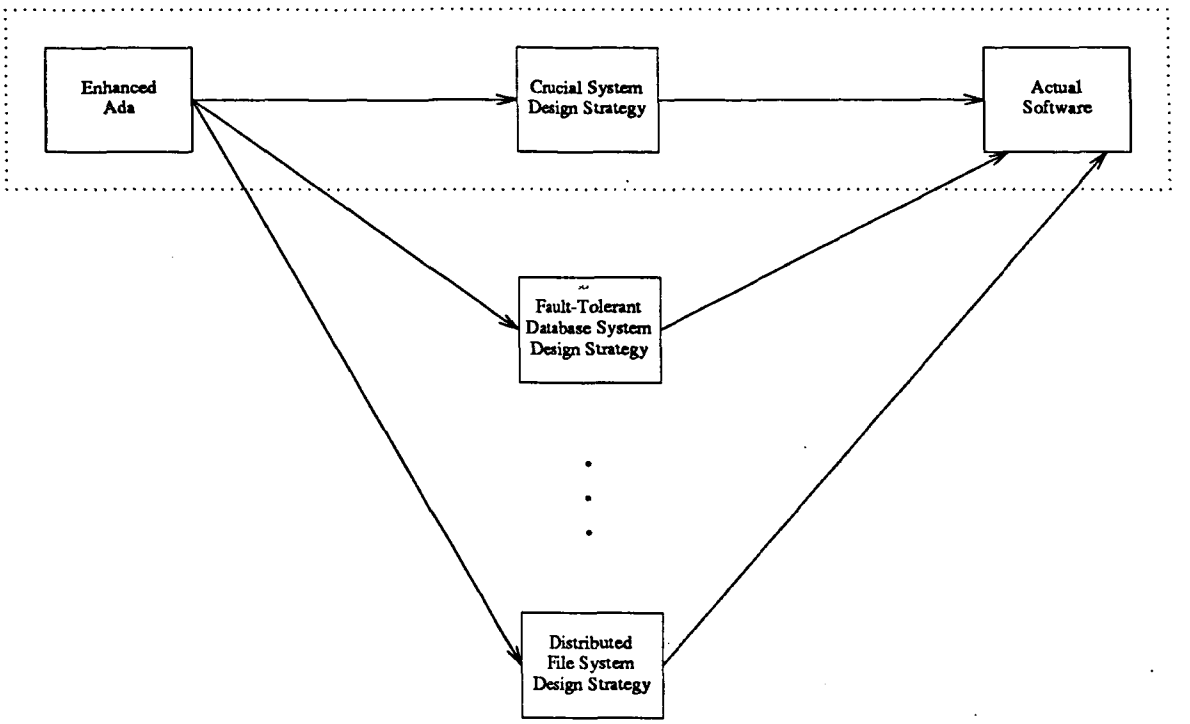

Figure 2 - Research Concerns

construction of other kinds of systems. The nonrestrictive partitioning philosophy of APPL is maximally supportive of the widest range of distributed application programming.

As a consequence, APPL does not, pending the experimental evaluations, appear to produce the structurally deficient programs which result from the distribution semantics described in Knight[2].

The problems we have with APPL relate mainly to our concern with fault tolerance, so we will address those issues in the next subsection, "Failure Semantics." We are not concerned with the details of APPL; instead, we use the basic ideas presented in Cornhill [3] as a starting point for the specification of a hypothetical Ada partitioning language. We assume the existence

of that language throughout this research, and we do not address the details of its implementation. We use the acronym HAPPL, for Hypothetical Ada Program Partitioning Language, to refer to this postulated language, and, because its differences from APPL result primarily from failure-related issues, we describe it in the next section.

Failure Semantics

Cornhill [3] suggests that fault tolerance might be provided solely by judicious use of the APPL replication feature. We have noted previously two serious objections to this *transparent* approach:

(1) **No specification of service degradation.** Since the application program has no way of learning of a failure event, it cannot respond to loss of computing power with a corresponding degradation in the level of services provided. This means that the system must be oversupplied with computing power in order to meet its deadlines after a failure. Such excess may not be feasible for applications with stringent power or physical space constraints.

(2) **Excessive overhead.**

The APPL one-to-many mapping feature is not flexible enough to to serve adequately the entire class of fault-tolerant applications. In particular, many embedded systems have severe memory and execution speed constraints. Those systems cannot afford the overhead required by the automatic redundancy of the one-to-many mechanism; in fact, they cannot afford even to provide full service after a computer failure.

In order to tolerate hardware failure, embedded systems with stringent real-time requirements must

(1) incur minimal fault tolerance overhead,

(2) provide degraded but safe service after node failure, and

(3) respond rapidly to failure events.

Although in theory it does allow for rapid response to failure, the so-called *transparent* fault tolerance implemented by exclusive use of the one-to-many mechanism is costly and does not allow for degradation. A less costly approach is one which recognizes that the minimum overhead needed for provision of fault tolerance is an application-specific quantity. For example, consider an array of critical data that is updated periodically in a loop. Any automatic mechanism for achieving data redundancy would necessitate a separate invocation of an interprocessor communication protocol for each update to each array element. We believe such overhead to be prohibitively high. What is actually needed is for the array to be broadcast to other processors after all of its elements have been updated. Concern for efficiency therefore dictates the imposition of some degree of application program control over the process of preparing for failure.

In order to provide a different level of service following hardware failure, an application program must be aware of and must respond to the failure. Any scheme that purports to provide transparent fault tolerance capabilities with appropriate degradation simply by replicating some processes and data and not others is naive. The key problem is in communications between critical and non-critical portions of the program. If they do not provide for the possibility of failure, critical program parts will communicate unsuspectingly with parts that no longer exist. A further problem is that process replication promises to be hideously expensive. Making all copies of a dynamic, complex entity like a task behave as if there were only one requires substantial intra-copy communication. Given that overhead and degradation are such critical issues, we conclude that pure transparency, despite its advantages of portability and simplicity, is not

practical.

Unfortunately, since in a non-transparent approach failure is handled by the application program rather than by the run-time system, the rapidity of response is limited not only by the intrinsic constraints of the problem but also by the Ada language itself. In particular, Ada provides no mechanism (except for abort) by which one task may communicate unilaterally with another.

In Cornhill [3], it is suggested that, for some applications, it may be necessary for the application program and the execution support system to exchange information about failures. The raising of an exception in response to an attempt to reference an object that no longer exists because of a node failure is offered as such a means of exchange. For real-time systems, this is insufficient because many such systems will need to reconfigure immediately in order to meet real-time deadlines. It may be unacceptably slow for a particular process to delay its reconfiguration until it attempts to communicate with the failed machine.

In addition to supplementing Ada with a predefined exception for failed communication, we have suggested in our previous approach that the run-time system of each computer which remains after a failure call a predefined entry in a task devoted exclusively to the reconfiguration of that machine. This scheme has three major disadvantages.

First, the one-per-node mapping of reconfiguration tasks binds the implementation to a particular computer architecture. Since one of the major reasons for distributed computing is the ease with which incremental changes in computing power can be effected, dependency of the software on one hardware configuration reduces substantially the utility of the system.

Second, this strategy does not solve the aforementioned problem with rapid response. Because Ada does not allow one task to get the attention of another, some sort of polling mechanism must be used to propagate news of the failure to all processes. A polling rate which

would guarantee rapid response would also ensure inefficient operation. Alternatively, the abort statement could be used in conjunction with Ada's dynamic tasking facilities: instead of informing processes of failure when they ask, the algorithm would be to abort all tasks which "really need to know" and restart replacements for them. We would very much like to avoid resorting to such an ugly, makeshift solution.

Third, when a task is lost due to processor failure, its replacement must have a different name; consequently, all program units which made entry calls to the original task and need to make entry calls to the replacement must use the new name. It is appropriate to use a conditional statement which describes an actual change in behavior following a processor failure (see Figure 3a); it is confusing to use conditional statements in order to use a different name to make the same entry call (see Figure 3b).

We view the efficiency of non-transparent continuation and the capacity to specify degraded service to be absolutely essential to the applications with which we are concerned. The replication features of APPL may have some utility (easy data redundancy, hot standby processes), but they are in general too expensive for systems with tight real-time, memory, and physical space constraints. However, the specific approach in Knight [2] leads to programs which suffer from

```
if BEFORE_FAIL then
  SERVER.CALL;
else
  DO_SOMETHING_ELSE;
endif;
```

(a)

```
if BEFORE_FAIL then
      SERVER.CALL;
    else
      ALT_SERVER.CALL;
endif;
```

(b)

Figure 3 - Replacement Naming Problem

(1) poor modifiability,

(2) slow or inelegant failure response mechanisms, and

(3) cumbersome naming of replacement tasks.

Our solutions to these problems are, in order,

(1) to use an APPL-like partitioning language to maximize distribution flexibility,

(2) to replace one-task-per-computer entry call failure signalling with predefined exceptions to be handled by all tasks which must know of a failure event,

(3) to use *resurrection*, a feature of our APPL-like language, to provide name replication without the costs of redundancy.

In the next subsection we describe the APPL-like language which we use to solve problems (1) and (3) above, and subsequently we define the the failure signalling mechanism that we use to solve problem (2).

Hypothetical Ada Program Partitioning Language (HAPPL)

Like APPL, HAPPL allows partitioning to be effected along task, package, subprogram, named block, and object boundaries. Complex objects may be partitioned along both indexed and selected component boundaries. Distribution of a particular construct C to a processor X means that, excepting the portion of C which is in turn explicitly distributed to another processor, all execution and storage requirements of C shall be fulfilled by X.

Unlike APPL, HAPPL requires partitioning to be static. The reason for this is to simplify the task of responding to a node failure. With static partitioning, the processes and data which are lost due to arbitrary failure of a given node are known at compile time, so the programmer does not need to handle the complexity of assessing arbitrary damage.

For most kinds of interprocessor communications, static partitioning has interesting ramifications. For example, the *meaning* of a procedure call across processor boundaries is different from that of a local call. Intuitively, we think of procedures as being executed *by* processes which are in turn executed by processors. In Ada, tasks execute subprograms, and processors execute tasks†. Given static partitioning, this paradigm cannot be applied to remote subprogram calls because it would imply migration of the calling task to the machine to which the subprogram was distributed. Instead, we must think of such calls as being executed by an *agent* on the remote machine. As its final act, the agent sends a message containing the results of the subprogram call to the actual calling task. The same idea works for remote data access, remote package instantiation, etc. From the perspective of the local task, remote access is, for each kind of communication, identical to local access. What makes agents useful, of course, is the resulting isolation of the calling process from the machine which executes the call. If an exception to handle failed interprocessor communication can be raised, failure of the called machine does not necessitate failure of the calling process. We define such an exception in the next section.

Besides static partitioning, the only other feature of HAPPL which does not exist in APPL is that of *resurrection*, which allows specification of one-to-many mappings which

† Even the main program, a subprogram, is executed by a task. In section 10.1 of the LRM, it says, "Each main program acts as if called by some environment task."

do not incur the overhead (nor provide the substantial functionality) of APPL one-to-many mappings. Basically, the stricture that copies behave as if they were actually a single construct is dropped. Instead, a resurrectable construct is a set of copies exactly one of which is active at a given time. When the machine executing the active copy fails, one of the dormant copies becomes active on one of the remaining computers. The application program is responsible, at the time of resurrection, for specifying the state of the newly activated construct. A resurrected data object, for example, initially will contain an undefined value, and access of that object prior to explicit post-failure initialization is erroneous.

As mentioned above, the whole point of resurrection is to replicate names without the high cost of process or data redundancy. It is intuitively clear that this can be done easily and usefully for data. It is not nearly so obvious what resurrection of a process might mean or how it might be useful.

To explore this issue, we need to make some general comments about the programming of recovery. The general approach to recovery is the following:

(1) **Detect failure.** Processor failure must be detected and communicated to the software on each of the remaining processors.

(2) **Assess damage.** It must be known what processes were running on the failed processor, what processes and processors remain, and what state the processes that remain are in

(3) **Select a response.** Information must be provided so that a sensible choice of a response can be made. This choice will normally depend on which processors and processes remain, but in many applications the choice will also depend on other

variables, and their values would have to be known.

(4) **Effect the response.** Once a response has been decided on, it must be possible to carry it out.

Failure Signalling

The only Ada construct which could allow all processes rapid access to the fact that a failure has occurred without binding software to particular hardware architectures is the predefined exception. Specifically, we designate an exception NODE_FAIL to be raised in every process which survives processor failure and has not suppressed the exception via the SUPPRESS_NODE_FAIL pragma to be defined below. That pragma may appear in task or task type specifications. In tasks whose specifications contain this pragma, the NODE_FAIL exception will not be raised.

For those processes which must continue processing after a node failure but do not need to be interrupted, the exception COMM_FAIL is raised in every process which is suspended due to an attempt to communicate with a failed node. This exception is only raised if the process has suppressed NODE_FAIL or has initiated the communication after processing the NODE_FAIL exception.

As described briefly above, a resurrection feature is available through HAPPL. Any partitionable Ada construct may be defined, via HAPPL, to be *resurrectable.* If a resurrectable construct is on a machine which fails, the run-time system is obligated to instantiate a new construct of the same name on a specified node. Only one relationship is guaranteed to exist between failed and resurrected constructs of the same name and type: those constructs which execute as processes (tasks and the main program) shall execute the same NODE_FAIL exception handler as would the failed construct had its processor not failed. The effect of process constructs which suppress the NODE_FAIL exception and are defined to be resurrectable after

failure is undefined after such resurrection. Programs which contain such constructs are defined to be erroneous.

The NODE_FAIL exception allows critical processes to respond immediately to failure. Those processes which do not need to respond to the failure event itself are protected by the COMM_FAIL exception. Such processes use the SUPPRESS_NODE_FAIL pragma to remain unaffected by the loss of a processor. The resurrection feature eliminates the need for programmed redirection of communication in program units which access remote portions of the program. Because a resurrected process or data object has the same name as the failed unit, program units which access it need not know of its failure. Compared to the APPL one-to-many mapping feature, resurrection incurs little overhead and is therefore more appropriate for general use in real-time systems.

SECTION 4

EVALUATION CRITERIA

Our approach to evaluation in this research is to attempt to construct software in Ada that meets the specifications of a real-world system and also tolerates hardware failures. Analysis is performed during and after construction to determine the success or otherwise of the non-transparent approach. In order to judge systematically the quality of the solution, and hence the value of the design method itself to the chosen example application, we have defined two major criteria which must be satisfied by implementations of the application: they must be easy to construct and maintain, and they must be sufficiently efficient to satisfy the constraints of the application. We maintain that any implementation which satisfies the two criteria is indeed feasible and therefore encouraging of further research. These criteria are intended to be general standards by which any non-transparent approach can be judged in the context of any application required to tolerate processor failure.

The areas of concern are program engineering and run-time overhead. We do not at this time have a full-scale implementation for real-time testing, so our estimation of overhead is purely analytic and therefore somewhat suspect. Because of this limitation, we will in our evaluations focus most intensively on the program development and maintenance aspects of the problem; however, our bias does not reflect comparative disregard for execution-time efficiency. The two criteria must be deemed equally important, because both must be satisfied in order to judge the non-transparent method to be feasible.

Development and Maintenance Issues

The extent to which fault-tolerant programs fail stylistically is of great concern, since the chosen fault-tolerance approach places much of the burden of recovery on the programmer, and the effect of that onus on a realistic program structure is unknown at this time. In order for fault tolerance to be feasible for Ada programs, those programs must be well-structured.

The fundamental goal of the modern software engineering and language design strategies which inspired Ada is flexibility. People want three kinds of flexibility:

(1) **Software.** When requirements change, programs should be easy to alter.

(2) **Personnel.** Software development and maintenance should not be hampered unduly by personnel changes. Software should be readily understandable.

(3) **Target architecture.** Software should be maximally independent of the computing system which executes it.

Regarding (1) above, we expect that a fault-tolerant program will be more difficult to change than a program which implements the same "normal" requirements but does not continue after a node failure. Because fault-tolerant programs must contain code to specify both pre- and post-failure services, changes to the requirements often will result in changes both to the primary and to the alternate software. This is not a serious problem, because a fault-tolerant program is a substantially more useful (and correspondingly more complex) product than one which is not. At the same time, if continuation causes a program to be more difficult to modify than one would expect (given the increased functionality of the fault-tolerant program), a substantial liability must be ascribed to the continuation facilities.

Regarding (2) above, a similar argument holds. Only if new personnel found understanding a fault-tolerant program to be more difficult than understanding a non-fault-tolerant program *of equal inherent complexity* should continuation be viewed as a detriment to comprehensibility.

The third kind of flexibility, independence from the underlying hardware architecture, is especially important for distributed systems†. A major reason for using distributed systems at all is to allow for efficient implementations of parallel algorithms. A number of competing issues arise:

(1) minimization of inter-node traffic,

(2) maximal exploitation of concurrency which is intrinsic to the algorithm, and

(3) even distribution of the computing load among the nodes.

Because of the obvious complexity, optimal partitioning is extremely difficult to achieve. The multi-program paradigm for distributed software requires that the basic partitioning decisions be made near the start of the development process, so system designers must resort to educated guesses. In the event of a mistake, functions must be shuffled between programs. This can be tricky, since the programs do not necessarily provide interchangeable software environments to the functions which may be subject to redistribution.

A major reason for using a single program on a distributed target is to be able to delay partitioning decisions until the software has been constructed. A function that is written as a distributable entity can be moved to a different processor quite simply if the movement is within a single program. Theoretically, all that is required is a change to the distribution specification, recompilation, and relinking. In principle, then, an accurate, empirical approach to program

†Portability, a major concern for *all* software systems, falls in this category of flexibility, but, as we stated earlier, we do not examine in this research the possible effects of continuation on portability.

partitioning can be employed without incurring massive development overhead.

Another area of flexibility provided by distributed systems is the ease of incremental change in computing performance. If, during development, it is discovered that the estimate of required performance is incorrect, processors can be added or deleted as needed. Adding or removing a processor to or from a system incorporating non-transparent fault tolerance amounts to requiring the movement of several functions at once from one processor to another. It is important to ensure that non-transparent continuation does not reduce the flexibility of distributed systems to the point where useful flexibility is lost.

Loss of the ability to move distributable entities from one node to another due to the need to survive hardware failure must be viewed as a serious negative in our evaluation of the proposed semantics.

A final concern at development time, is the possibility that the inclusion of non-transparent continuation may so distort the desired form for a program that properties of programs in the software engineering sense, such as good modularity, information hiding, etc., might be lost. These are difficult properties to quantify and are, to a large extent, subjective, but any indication that these properties must be sacrificed for recovery would be serious.

In general, the program engineering criterion is subjective and therefore an unattractive metric, but we believe it to be critically important. In an age when the Federal Government targets millions of lines of Ada code for crucial systems, we can ill-afford to neglect the very principles and techniques which make such a goal conceivable. Of special interest will be the task distribution flexibility, the overall complexity, and the ease with which modularization and object isolation techniques can be applied to the problem.

Run-Time Issues

At execution time, the major concern is overhead. The resources used by non-transparent continuation must not reduce overall performance to the point where real-time deadlines cannot be met. Both the application program and the run-time support systems (distributed one per node) will incur overhead in the areas of communications bus traffic, processor cycles, and memory.

For the application program, the following are likely sources of overhead:

(1) bus traffic due to the need to distribute crucial data items to other machines in order to provide for recovery,

(2) processor cycles used for memory copy and I/O operations associated with data distribution and for the execution of conditional and select statements needed to choose between primary and alternate software,

(3) storage of crucial data items, extra conditional and select statements, alternate service software, reconfiguration software, data distribution software, code for resurrectable processes, and possible post-failure incarnations of resurrectable data items.

For the run-time support systems, the following are sources of overhead:

(1) bus traffic due to interprocessor communication for failure detection and for implementation of the semantics of resurrectable processes,

(2) processor cycles used for maintenance of the message log [2] (to generate appropriate COMMUNICATION_FAILURE exceptions after a node failure), for failure detection, and for implementation of the semantics of resurrectable processes, and

(3) storage of the message log and the portion of the support system software needed specifically to implement failure semantics,

How much overhead is too much? The value of continuation depends upon the application, so each case should be considered independently; however, if the run-time processor cycle overhead were to approach fifty percent, we would look for another solution to the reliability problem. Memory overhead is not nearly as sensitive an issue, since the address space of most modern computers is huge, and memory components are quite inexpensive. Similarly, the level of extra bus traffic is not particularly important, since modern fiber-optic buses provide bandwidth far in excess of that needed by current systems.

An issue related to overhead is response time. It must be possible for recovery to take place fast enough following the loss of a processor to ensure that the equipment being controlled does not suffer from a lack of service. There is a trade-off between overhead and response time. More time can be made available for the response if crucial data are distributed more frequently or if more data are deemed to be crucial. Also, the execution of certain alternate facilities even when a failure has not occurred may decrease the time needed to make those facilities operational after a failure.

SECTION 5

DESIGN ISSUES

The semantic enhancements proposed in section 3 provide the necessary tools for the programming of crucial, real-time, distributed systems. This section addresses the question of how those tools may be used to construct programs which measure up to modern software engineering standards. Two requirements are peculiar to fault-tolerant systems: ensuring the availability of crucial data after failure and configuring the post-failure system to provide a level of service which reflects the loss of computing power. In the remainder of this section, we show how each kind of service can be provided by the proposed semantics.

Maintaining the System State

For most applications, in order to recover from a failure, some information about the state of the software on the failed machine must be available. Obviously, to withstand one processor failure, the state data must be stored on at least two nodes. For an automatic aircraft control system, for example, the current position of the aircraft, the status of the sensors, and historical information for the navigation integration filters are essential to timely recovery. It is also important that the information available to the application program represent an internally consistent state. For example, a flight plan which is a composite of two sets of contradictory pilot specifications is useless; an out-of-date but consistent flight plan could be quite useful.

One solution might be to execute periodically a highest-priority process which collects all crucial data and stores it on two machines. Because Ada priorities are relevant across nodes only when inter-node communication occurs, processes on machines other than the one which executes the collection process may alter crucial items while the system snapshot is being taken.

Also, processes on the machine which executes the snapshot process could run while the snapshot process awaits response from another machine. Both cases violate the stricture that the snapshot be internally consistent. A rather clumsy way to solve this problem is to have the collection process start infinitely looping do-nothing processes on all machines. These processes would execute at a priority just beneath that of the snapshot process, so no "real" processes other than the data collector would execute until the do-nothing processes were aborted. This is a costly approach, since no useful work may proceed in parallel with the data collection; however, it is simple and unsusceptible to error. Since this strategy will work for any application, we call it the *universal* approach to maintaining the system state for recovery. The only impact data distribution has on the design of the rest of the program is in the visibility of crucial data--even strictly local items must be exported from the packages in which they are used.

For applications which cannot tolerate the inefficiency of stopping all processes during crucial data acquisition, a variety of *contextual* techniques can be used. One would be to analyze the crucial data for specific time interdependency groups and to access those groups via critical sections only. The flight plan, for example, must be consistent only within itself--the time at which the sensor statuses were saved is irrelevant. Another technique would be to use the partial ordering of events which is apparent even in distributed, real-time software to decide whether or not particular crucial data should be protected by critical sections. Still another approach would increase efficiency by updating only those items which have changed or by updating items at different rates.

We favor a hybrid approach in which no other work is done during the period needed for the snapshot and also in which distribution of data is performed at rates appropriate to the individual items and only when they have been updated.

Specifying the Post-Failure Level of Service

We have identified four ways in which one might choose to reduce safely the level of service provided by a program:

(1) Eliminate non-critical functions entirely.

(2) Use faster but less accurate algorithms wherever the loss of accuracy is safe.

(3) Increase the length of one or more of the real-time cycles whereever the slower service is safe.

(4) Execute functions less frequently within a particular real-time frame whereever the loss of speed or accuracy is safe.

The two major advantages of non-transparent over transparent continuation are (a) control of overhead due to continuation and (b) the ability to specify a new level of service which is appropriate to the remaining computing power yet still safe. Obviously, the key assumption behind (b) is that real-time systems typically provide a level of service beyond that which is sufficient to avoid catastrophe. For (1) above to be applicable, it must be possible to divide a given application into critical and non-critical functions of which the non-critical functions are a significant percentage of the whole.

For (2), (3), or (4) to be applicable, there must be a substantial difference between the minimum real-time constraints which ensure safety and those which are met by the computing system. Such a difference can arise in three ways. First, a time slice in which frame overruns are not disastrous may be incorporated into a system in order to provide a margin of safety in the case of difficulties unforeseen by the software designer. Real-time systems are quite complex, and it seems unlikely that people typically will build them to meet their deadlines at the limits of safe

operation. Second, an intrinsic difference may exist between *desirable* and *acceptable* constraints. For instance, in an aircraft control system, a slower rate of computation of the positions of the control surfaces will not cause the aircraft to plummet to the ground; instead, the trajectory of the aircraft will become less smooth, and the passengers will be correspondingly less comfortable. Certainly, in the emergency situation created by loss of computing resources, comfort may justifiably be sacrificed in the interest of survival. Finally, the discrepancy may result from a common psychological phenomenon. From our own experiences, we note that people tend to use so-called "round" numbers whenever possible. For example, in the aircraft control system with which we are familiar the designers settled on a cycle length of ten milliseconds. It seems entirely possible, given what we suspect is a common human trait, that the ideal cycle length might actually be eleven, twelve, or even thirteen milliseconds. Further, it is unlikely that the ideal cycle length is less than ten milliseconds, since the aircraft would not fly well if the computing system which controlled it did not perform its cyclical tasks fast enough.

REFERENCES

(1) Reference Manual For The Ada Programming Language, U.S. Department of Defense, 1983.

(2) J.C. Knight and J.I.A. 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.

(3) D. Cornhill, "A Survivable Distributed Computing System For Embedded Applications Programs Written In Ada", *ACM Ada Letters*, Vol. 3, pp. 79-87, December 1983.

APPENDIX

REPORT LIST

The following is a list of papers and reports, other than progress reports, prepared under this grant.

(1) Knight, J.C. and J.I.A. Urquhart, "Fault-Tolerant Distributed Systems Using Ada", Proceedings of the *AIAA Computers in Aerospace Conference*, October 1983, Hartford, CT.

(2) Knight, J.C. and J.I.A. Urquhart, "The Implementation And Use Of Ada On Fault-Tolerant Distributed Systems", *Ada LETTERS*, Vol. 4 No. 3 November 1984.

(3) Knight, J.C. and J.I.A. Urquhart, "On The Implementation and Use of Ada on Fault-Tolerant Distributed Systems", *IEEE Transactions on Software Engineering*. May, 1987.

(4) Knight J.C. and S.T. Gregory, "A Testbed for Evaluating Fault-Tolerant Distributed Systems", Digest of Papers FTCS-14: *Fourteenth Annual Symposium on Fault-Tolerant Computing*, June 1984, Orlando, FL.

(5) Knight J.C. and S.T. Gregory, "A New Linguistic Approach To Backward Error Recovery", Digest of Papers FTCS-15: *Fifteenth Annual Symposium on Fault-Tolerant Computing*, June 1985, Ann Arbor, MI.

(6) Gregory, S.T. and J.C. Knight, "Concurrent System Recovery" in *Resilient Computing Systems, Volume 2* edited by T. Anderson, Wiley, 1987.

(7) Knight, J.C. and M.E. Rouleau, "Analysis Of Ada For A Crucial Distributed Application", Proceedings of the Fifth National Conference On Ada Technology, Washington DC, March, 1987.

(8) Knight, J.C. and J.I.A. Urquhart, "Difficulties With Ada As A Language For Reliable Distributed Processing", Unpublished.

(9) Knight, J.C. and J.I.A. Urquhart, "Programming Language Requirements For Distributed Real-Time Systems Which Tolerate Processor Failure", Unpublished.

DISTRIBUTION LIST

Copy No.	
1 - 3	National Aeronautics and Space Administration Langley Research Center Hampton, Virginia 23665 Attention: Mr. C. Michael Holloway ISD M/S 125
4 - 5*	NASA Scientific and Technical Information Facility P.O. Box 8757 Baltimore/Washington International Airport Baltimore, Maryland 21240
6 - 7	J. C. Knight, CS
8	R. P. Cook, CS
9 - 10	E. H. Pancake, Clark Hall
11	SEAS Publications Files

*1 reproducible copy

0240:ald:DW209R

UNIVERSITY OF VIRGINIA
School of Engineering and Applied Science

The University of Virginia's School of Engineering and Applied Science has an undergraduate enrollment of approximately 1,500 students with a graduate enrollment of approximately 560. There are 150 faculty members, a majority of whom conduct research in addition to teaching.

Research is a vital part of the educational program and interests parallel academic specialties. These range from the classical engineering disciplines of Chemical, Civil, Electrical, and Mechanical and Aerospace to newer, more specialized fields of Biomedical Engineering, Systems Engineering, Materials Science, Nuclear Engineering and Engineering Physics, Applied Mathematics and Computer Science. Within these disciplines there are well equipped laboratories for conducting highly specialized research. All departments offer the doctorate; Biomedical and Materials Science grant only graduate degrees. In addition, courses in the humanities are offered within the School.

The University of Virginia (which includes approximately 2,000 faculty and a total of full-time student enrollment of about 16,400), also offers professional degrees under the schools of Architecture, Law, Medicine, Nursing, Commerce, Business Administration, and Education. In addition, the College of Arts and Sciences houses departments of Mathematics, Physics, Chemistry and others relevant to the engineering research program. The School of Engineering and Applied Science is an integral part of this University community which provides opportunities for interdisciplinary work in pursuit of the basic goals of education, research, and public service.