ANALYSIS OF Ada FOR A CRUCIAL DISTRIBUTED APPLICATION

John C. Knight

Marc E. Rouleau

Computer Science Report No. TR-87-02
February 6, 1987

Presented at the 5th National Conference on Ada Technology

ANALYSIS OF Ada† FOR A CRUCIAL DISTRIBUTED APPLICATION‡

John C. Knight Marc E. Rouleau

Department of Computer Science
University of Virginia
Charlottesville, Virginia, 22903

SUMMARY

Ada was designed for the programming of embedded systems, and has many characteristics intended to promote the development of reliable software. Many embedded systems are distributed, and an important characteristic of such systems is the ability to continue to provide adequate though perhaps degraded service after loss of a processing node. We are concerned with the issues that arise when critical distributed systems are programmed in Ada. We have shown previously that numerous aspects of Ada make its use on distributed systems problematic if processor failures have to be tolerated. The issues are not raised from efforts to implement the language but from the lack of semantics defining the state of an Ada program when a processor is lost. We have suggested appropriate semantic enhancements to Ada and have described the support system required to implement these semantics. We are evaluating and refining this approach by applying it to a large-scale aerospace system. The application is the resident software for a modified Boeing 737 transport equipped with a fully automated, digital, control system. The framework for a fault-tolerant version of the complete system has been constructed, and certain critical functions have been programmed in their entirety. In this paper we present a summary of the application, details of its implementation in Ada, and a preliminary evaluation of the utility of Ada in this context.

---

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

‡ This work was supported in part by NASA grant NAG1-260

INTRODUCTION

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. Success in this endeavor will make these systems *more* reliable than single-processor architectures. More importantly, distributed systems can, if necessary, provide degraded service following the failure thereby allowing either reduced service or a timely, controlled shutdown. In real-time control applications requiring very high reliability, such as are found in the defense and aerospace areas, this can be extremely important.

Ada[1,2] has been designed for such critical, real-time applications and for the development of programs distributed over multiple computers. Unfortunately, it has been shown to be deficient in this area[3]. In fact, the Ada definition ignores the problem completely and implies that the Ada machine does not fail. With this in mind, some researchers have proposed a *transparent*[4,5] approach to recovery 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. Operation of the system before and after node failure would be identical. The burdens of recovery and of the preparation needed for that recovery are placed upon the execution-time environment.

Knight and Urquhart[3] 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, or *safe*[6], 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 operation 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.

Previous research has shown that the non-transparent method is *theoretically* feasible. A distributed testbed containing an execution-time system which provides the necessary facilities for non-transparent recovery has been constructed[7]. A simple distributed Ada program, written according to certain guidelines, can be run on the testbed, and arbitrary failure of one of the nodes will initiate reconfiguration and the return to "acceptable" operation (lacking one computer).

The goal of the research described here is to evaluate the *practical* value of non-transparent recovery as proposed by Knight and Urquhart by analyzing their feasibility in the context of one realistic application. The application is NASA's Advanced Transport OPerating System (ATOPS), an aircraft computing system capable of performing all flight tasks from take-off through landing. While the resultant data point will provide, in and of itself, little persuasive evidence for or against the proposed non-transparent method, the associated analysis should provide useful insight into the issues relevant to hardware-fault-tolerant Ada distributed systems.

This paper is organized as follows. In the next section we summarize the issues surrounding the use of Ada on fault-tolerant distributed systems. The application being examined in this experiment is then described, and the design of the experiment, including the criteria used in evaluation, is then presented. The preliminary analysis of the experiment is then discussed and finally a summary and conclusions are presented.

Ada SEMANTICS AND NON-TRANSPARENT CONTINUATION

In this section we summarize the difficulties with Ada semantics and the approach to non-

transparent continuation suggested by Knight and Urquhart. For more details see reference 3.

Deficiencies In Ada Semantics

There are two categories of semantic deficiency in Ada. First, the definition of Ada lacks any clear definition of the meaning of a distributed Ada program, i.e., there are no *distribution semantics*. Specification of the basic units of distribution, whether they be individual statements, tasks, packages, or anything else, is omitted as is the detailed meaning of such a program. This is surprising since representation clauses are available to control many other implementation areas in great detail.

The lack of distribution semantics is not a simple problem to solve. It is not sufficient merely to define what can be distributed. It is also necessary to define exactly what distribution of an object will mean. For example, if a task can be distributed, it is essential that the meaning of the distribution include details of the location of the associated code and data. A syntax to control distribution is also required, of course.

The second semantic deficiency is the lack of definition of the meaning of a program following a hardware failure. An Ada program operating on a distributed target is obviously going to be damaged by the loss of a node from the system. The extent of the damage must be precisely defined since it goes beyond the loss of the software on the failed node and affects the software on nodes that survive failure. For example, if two tasks are engaged in a rendezvous when a failure occurs, the caller would be permanently suspended if the server was lost since the rendezvous would never end and the caller could not distinguish this situation from slow service by the server.

Another area where damage to tasks surviving failure can occur is the possibility of loss of context. If a nested task survives failure but the surrounding task is lost, the nested task may lose

part of its context (the local variables for the surrounding task) and cannot be accessed since no remaining undamaged tasks can know its name.

The *failure semantics* of a programming language define precisely the state of the software that remains following the loss of a processor at an arbitrary point. Failure semantics for Ada must be defined if the software that remains is to be in a form that can be predicted accurately during design. This is necessary if some form of service is to be provided reliably after a failure.

Non-Transparent Continuation

Distribution and failure semantics for Ada that involve no syntactic changes other than the definition of a pragma have been defined for Ada[3]. The major component of distribution in the approach is the task. A program is structured as a set of tasks with a main program that consists solely of a null statement. The tasks nested within the main program can be distributed as required using a pragma. Tasks nested at lower levels are allocated to the processor of the surrounding task(s).

Failure semantics are defined to be equivalent to abort semantics. Where a task is lost through hardware failure, it is assumed that it was in fact aborted and the effect on the remaining program is therefore well defined. It is necessary to supplement the usual Ada execution-time support system in order to provide these semantics. Continuing with the example of the broken rendezvous used above, in that case it is assumed that the lost task was aborted and the support system will be required to raise a tasking error exception in the calling task.

Each processor has a *reconfiguration* task associated with it and this task receives a call to a predefined entry when a processor failure occurs. The support system is required to monitor processor health and signal the loss of of a processor to those that remain by making this entry call when necessary. The reconfiguration task executes code that takes care of the needs of the

processor on which it resides. Alternate tasks for those that have been lost reside on remaining processors and are activated by the reconfiguration task using entry calls.

A necessity in any form of continuation is the availability of data that survives the failure. Many components of an application update local data that must be available on each real-time cycle. To recover such a function, a consistent copy of the local data must be available on the processor that executes the alternate software. In a distributed system the reliable distribution of such data can be achieved with a two-phase commit protocol[8,9]. In the proposed non-transparent continuation, the data to be made consistent is determined by the programmer and the times when the copies are updated are the programmer's responsibility. Data distribution is achieved by rendezvous with a *data consistency* task that implements the two-phase commit protocol.

All inter-task communication takes place using the features provided by Ada whether this is inter- or intra-machine communication. This provides uniformity for the programmer and allows compile-time checking. Following failure, however, services will be resumed by alternate tasks to replace those lost by failure and these tasks will have different names. This means that all inter-machine communication will have to be programmed with explicit selection of the entry to be used. This has a substantial effect on program structure and hence on performance.

EXPERIMENT DESIGN

Our approach in this research is to attempt to construct software in Ada to meet the specifications of the ATOPS system and to incorporate tolerance to hardware failures also. Analysis is performed during and after construction to determine the success or otherwise of the non-transparent approach.

No single conclusion about the utility of non-transparent continuation can be drawn from this research since what constitutes a serious limitation in one context might not be in another.

There are several distinct criteria that must be used in the evaluation; each criteria affects the overall determination of the success of the method in any given application. These criteria fall into three broad categories - development-time issues, execution-time issues, and the effect of different target architectures. In this section we discuss the criteria we are using and the reasons for their use.

Development-Time Issues

Non-transparent continuation allows the programmer to determine the details of recovery following failure. The reconfiguration software and the software for alternate services must be prepared during development but we do not consider this software as overhead since it constitutes the implementation of a substantially more useful application than an implementation with no recovery. However, its volume can be measured and memory space must be available for it.

It is very desirable to delay decisions about binding functions to processors as late as possible in the development process. A major reason for using a single program on a distributed target is flexibility. A function that is written as a distributable entity, say a task, can be moved to a different processor quite simply if the task is merely part of a single program. All that is required, in principle, is a change to the distribution directive, recompilation, and relinking. Non-transparent continuation reduces this flexibility since, if the function being moved has to survive processor failure, alternate software must be provided. Thus, moving a function requires not only the possible movement of the alternate software, but also the revision of the reconfiguration tasks, changes to the use of the data distribution tasks, and possibly changes to the other software. For example, some communication may have to be modified to be prepared for redirection, and other communication may have to be modified to remove the ability for redirection.

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.

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.

Execution-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. Areas where overhead will be incurred are:

(1) communications bus traffic,

(2) processor overhead used in data distribution,

(3) processor overhead used in the implementation of failure semantics,

(4) processor overhead used by the application to determine select between primary and alternate software, and

(5) memory space required to support alternate software.

An issue related to overhead is response time. It must 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. The reconfiguration task must be started, must perform its services, and alternates must be started sufficiently quickly that real-time deadlines are not missed.

Target Architecture

All of the above criteria are affected in practice by the target architecture. The key elements of the architecture are:

(1) the number of processors provided,

(2) the types of the processors and the number of each of the different types,

(3) the relative performance levels of the processors, and

(4) the sizes of the supplied memories.

Clearly, if a system is operating with all resources fully used, it will be impossible to maintain all services following failure. In fact, if all available memory is used in providing the original services, no fault tolerance will be possible since there will be no memory space for alternate software or reconfiguration software.

In some systems, specialized processors are provided to allow processor organizations that are tailored to specific needs of the application. For example, an array processor or a fast-fourier-transform processor might be provided to enhance performance. Loss of this type of equipment has a different effect on a distributed system than loss of a general-purpose processor because it is unlikely that the remaining equipment will be able to provide service at the speed

required. Some compromise will be necessary. Similarly, it is unlikely that a specialized processor will be able to take over the services lost when a general-purpose processor is lost.

In this research we have selected a small number of architectures that we feel are typical and that can support the ATOPS application. We are investigating, separately, architectures involving two, three, and four homogeneous processors. We assume that there is sufficient memory on each to accommodate the necessary redundant software but that in each case almost all of the available computing resources are used by the original application. Thus each analysis is supplemented by an attempt to add a processor to each target being investigated.

APPLICATION DESCRIPTION

The application we are analyzing in this research is an experimental aircraft navigation and control system capable of performing automatically all flight tasks from take-off through landing. The production system operates on a modified Boeing 737 in a flight controls research program at NASA Langley Research Center. The operational software is written mostly in HAL/S[10]. In principle, failure of such a computing system could entail loss of control of the aircraft, so the application is a candidate for fault tolerance. In practice, the HAL/S system operates with a safety pilot and a complete set of backup controls that operate conventionally.

Our analysis is theoretical and is based on rewriting only parts of the system in Ada. The software we have written is not intended to be used operationally, and none could ever be used on the actual aircraft. Our intention is to use the operational system as a realistic example so as to ensure that the analysis we perform takes all the real-world problems into account. We have assumed certain required functions are critical even though for this flight control application they might not be. Such functions would be critical in other applications and the reason for making these assumptions is to make the analysis more likely to apply generally.

The remainder of this section contains a complete but superficial description of the operation of the aircraft computing system. More detail can be found in reference 11.

Inputs

All inputs to the system fall into one of three general categories. First, aircraft sensors provide the software with situational information. These devices measure such quantities as wind speed, vehicle acceleration, flap positions, roll, pitch, and yaw. Due to their unreliability, all sensors are duplicated or triplicated. A major element of the system's software is management of the redundant sensors to ensure that computations are only performed with good sensors.

Ground-based radio and microwave navigation aids make up the second category of input. During normal flight, angle and distance information from radio beacons at known locations allows the software to compute aircraft position accurately. The computed position then serves as a periodic correction in the integration of acceleration into velocity and position. In the vicinities of certain airports, faster and highly accurate position information is available from Microwave Landing System (MLS) transmitters. The concomitant improvement in the accuracy of the location information allows fully automated landings.

The third and final source of input is the cockpit. An enhanced joystick, known as a *brolly handle*, permits the pilot to fly the plane in a conventional fashion through the computer system. The flight control panel allows the pilot to select the desired level of automatic flight assistance. This panel also allows him to specify digitally the desired speed, altitude, flight path angle, and track angle. The navigation control display keyboard is the medium by which the pilot can manage the navigation functions of the system. He can use the keyboard to request general information about airports and navigational aids, and to enter or to modify the flight plan. Two other control panels allow the pilot to select display formats for the two horizontal situation

indicators. A final control panel controls the display format of the attitude director indicator.

Outputs

All outputs from the system fall in one of two major categories. The first category is the effector outputs that control flight of the aircraft. These outputs include commands to the ailerons, delta elevators, rudder, and throttle.

The second category of system output is feedback to the cockpit. The lighting configuration of the flight control panel informs the pilot of the level of automatic flight assistance currently being offered. Digital readouts on this panel tell him the speed, altitude, flight path angle, and track angle. These values are either current or desired as indicated by the lighting configuration of the panel. Character outputs to the navigation control display screen provide appropriate feedback to the keyboard requests of the pilot. Information sent to the two horizontal situation indicators allows the pilot to observe the progress of the aircraft as compared to the flight plan. Data output to the attitude director indicator permit the pilot to monitor the orientation of the aircraft in space.

Functional Units

The application can be divided into four major functional units. Navigation computes the *situation* of the aircraft. The concept of situation includes such measurements as position, velocity, acceleration, roll, pitch, and yaw. The navigation unit computes these quantities from flight sensor readings and ground aid transmissions.

Several sensor systems with overlapping functions are used by the navigation unit. Microwave Landing System (MLS) transmitters, where available, provide the most accurate measurements of latitude, longitude, and position. When the aircraft is in the vicinity of these transmitters, integration of acceleration measurements from the Inertial Navigation System (INS)

gyros on board the plane use the MLS-derived position as a correction. A second less flexible ground-based landing aid, the Instrument Landing System (ILS), supplies the correction information near runways which are not equipped with MLS.

When the aircraft is not landing, data from radio beacons at known locations on the ground allow computation of the correction term. Other sensors measure true air speed, magnetic heading, magnetic variation, and barometric altitude, and these sensors comprise a correction and backup system to the other navigation sensor systems.

The second major function of the application is guidance. Guidance compares the aircraft position to the flight plan and generates appropriate steering and acceleration signals for the flight control laws. Horizontal, vertical, and time guidance are separately selectable on the flight control panel; however, vertical guidance is disabled until horizontal guidance is engaged, and time guidance cannot engage until vertical guidance has done so. Also necessary for engagement of a particular level of guidance is specification of a flight plan specific enough for that level. For example, time guidance cannot operate until arrival times are associated with each waypoint in the path.

Flight control is the third functional unit. Three control laws comprise the heart of the unit. The lateral control law computes the aileron and rudder commands, the vertical control law computes the delta elevator command, and the throttle control law computes the throttle command. Each control law uses inputs selected according to the flight mode. If no automatic guidance mode is engaged, the primary inputs will be from the brolly handle in the cockpit. If horizontal guidance is engaged, the lateral control law will use the lateral steering signal from guidance. If vertical guidance is engaged, the vertical control law will use the vertical steering signal from guidance. If time guidance is engaged, the throttle control law will use the acceleration signal from guidance.

The fourth and final function of the application is display. The display function provides formatted data to the two horizontal situation indicators and the attitude director indicator for display to the pilot. The three control panels associated with the indicators indicate the formats and contents of the display output data. Navigation and guidance results are the inputs to the display unit.

Timing

Each of the functions described above operates at one of three speeds. Certain critical inputs and outputs are handled every ten milliseconds. Most of the computations are performed every fifty milliseconds. Finally, some computations, especially those driven by inputs from a human source, are performed essentially as background functions whenever the ten and fifty millisecond computations are complete.

PRELIMINARY ANALYSIS

In our preliminary analysis, we have concentrated on the target architecture that is presently being used by the operational ATOPS system. This is a dual processor configuration in which flight-management and flight-control (FM/FC) functions reside on one computer, and sensor management and display functions reside on the other. This original partitioning of functions was determined by the desire for co-location of similar functions. In practice, it allows the existing operational system to meet its real-time deadlines. For identification, we refer to the two computers as the "FM/FC" and "Displays" computers respectively.

We have maintained this general partitioning and added recovery to it. This approach has the virtue of determining the feasibility of adding recovery to an existing implementation as well as evaluating the approach using the criteria outlined in section 3. Modification of an existing operational design represents a substantial initial test since, if it can be done successfully, it

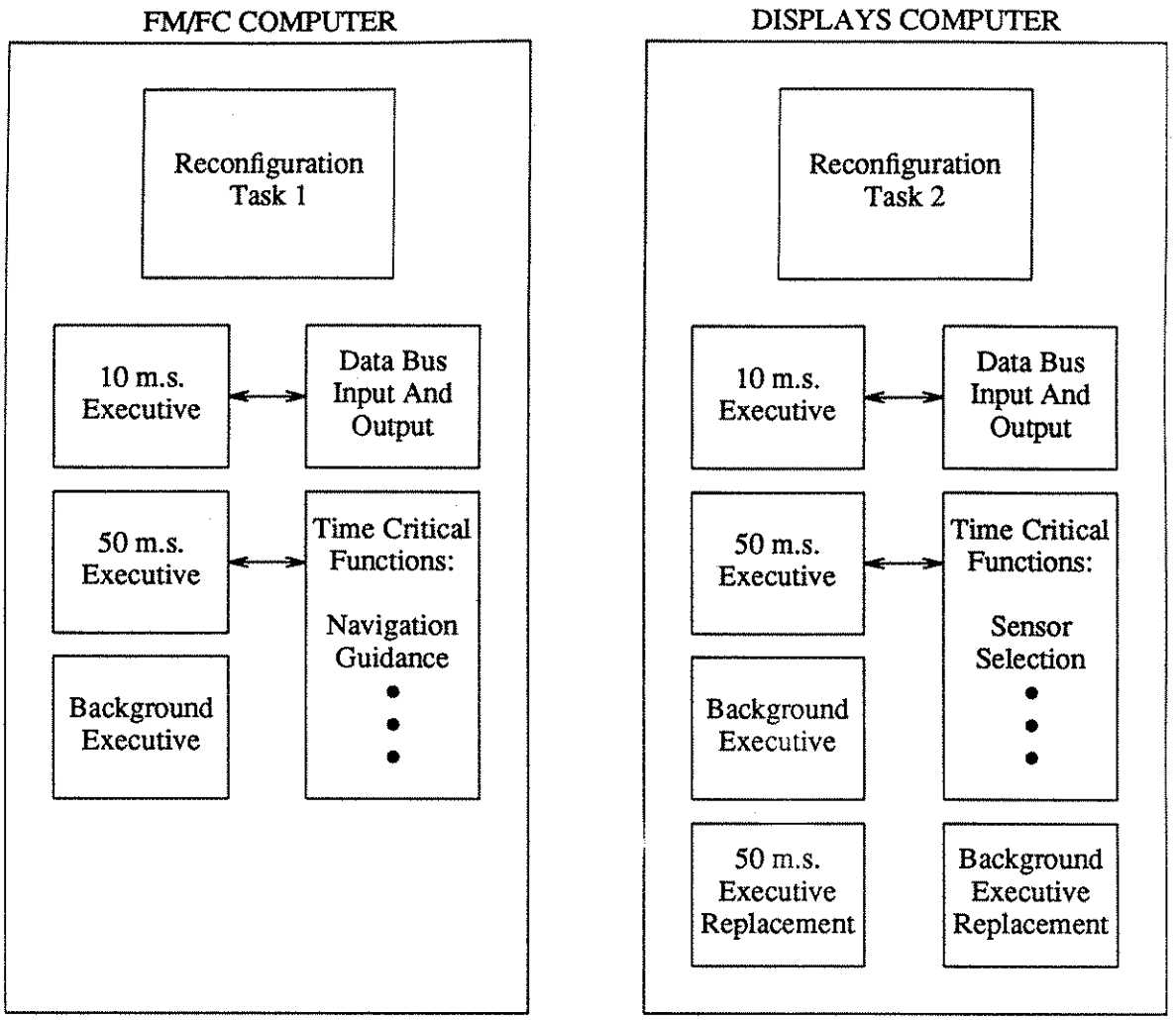

Fig. 1 - Software Structure

indicates that existing systems need not be discarded in order to take advantage of non-transparent continuation.

In a paper of this length it is not possible to document all the details of this analysis and so we present only a summary. The overall software structure is described followed by a more detailed examination of one component of the system. We then summarize the results of applying the various evaluation criteria.

Software Structure

The software architecture that we have designed for this target is shown in figure 1. During normal operation, each processor executes three tasks. One task on each processor executes every ten milliseconds and is synchronized by an external hardware clock. A second task on each processor executes every fifty milliseconds and is synchronized by entry calls from the associated ten-millisecond task. Finally, each processor has a background task that uses remaining processor time. Priorities are used to control the execution order of these tasks.

When both processors are operational, a block of data of between 200 and 3000 words, the length depending on whether certain new inputs are available, is transmitted from the FM/FC to the Displays computer every fifty milliseconds. The current sensor status is transmitted in the other direction at similar times.

The required recovery speed of the services being provided has determined the remainder of the software structure. The FM/FC functions must be resumed *very* quickly following failure. Despite the obvious inertia of a commercial air transport, certain control functions must be resumed in a few milliseconds. The functions provided by the Displays computer do not have to be recovered nearly so rapidly if it fails. These functions are interfacing with humans for the most part, either generating displays or accepting and processing inputs from keyboards and similar devices. Recovery within a few hundred milliseconds is adequate for these services.

The alternate software that takes over the services of the Displays computer when it fails is integrated into the FM/FC software. During each real-time frame, the FM/FC software checks the value of a flag that indicates status of the Displays computer. This flag is set by the FM/FC computer's reconfiguration task and is the only major action of that task. If the flag shows that the Displays computer is operational, the FM/FC software operates normally. If the computer has failed, different software is executed that provides reduced FM/FC service and skeleton display

and keyboard service. With this approach, it is possible that a considerable delay may occur between failure and resumption of display activity because the FM/FC software will only be aware of the failure after checking the flag. This may not occur until after extensive processing associated with FM/FC functions.

Since the FM/FC functions must be recovered rapidly, the approach of setting a flag following failure of the FM/FC computer and waiting for the Displays computer's software to check it is not sufficient. The software on the Displays computer cannot be designed like the FM/FC software just described. Our approach to ensuring very rapid recovery of the FM/FC functions is to locate on the Displays computer a skeleton version of parts of the FM/FC software. Also, complete replacements for the Display's computer's fifty millisecond and background executives reside on the Displays computer. There is no replacement for the Displays computer's ten-millisecond executive.

The reason for these replacement executives is to allow fifty-millisecond and background processing to be changed completely and very quickly on the Displays computer following failure of the FM/FC computer. These executives and all the alternate FM/FC software are normally idle. The executives contain entry definitions upon which they are normally suspended waiting for entry calls. Following failure, the replacement executives are started by entry calls from the reconfiguration task. The reconfiguration task also aborts the primary executives. The ten-millisecond executive is not replaced because its functions are, to a large extent, unchanged after failure. The reconfiguration task does set a flag so that this executive can make the necessary minor changes.

Since the reconfiguration task has the highest priority of all tasks on each machine, it is guaranteed to execute as and when required following failure. Thus the only delays between detection of the failure of the FM/FC computer and the resumption of FM/FC service on the Displays computer are (1) the time to start the reconfiguration task by generating an entry call, (2)

the time to start the replacement executives by entry calls from the reconfiguration task, and (3) the startup time of the replacement executives and associated computation functions. We expect these delays to be predicatable and small.

The skeleton FM/FC software is complete in that it can handle all the *critical* functions safely, although in some cases execution frequencies are reduced. Fortunately, it is not essential that all the control laws operate at the fastest real-time frame rate. Acceptable, although not perfect, performance is achieved at slower rates and so the replacement software operates at slower rates wherever possible.

Clearly, this is only one of many software structures that could be used. It is important to note, however, that the use of a skeletal replacement for the FM/FC software would not be feasible on any other target architecture. If the alternate software for the FM/FC functions had to be located on two or more processors, it would have to be partitioned in some way.

Microwave Landing System

As a detailed example of the way in which recovery might be provided for a critical function, we examine the Microwave Landing System software. The MLS system is an example of a function that we have viewed as critical although in a strict sense it is not. Failure of MLS service, even during landing, need not be disastrous provided that the pilot was informed and that he could fly the airplane manually. Taking over manually would be difficult if failure occurred close to the ground during an automatic landing but is possible. Recovery in this case could be limited to ensuring that sufficient functionality was available to allow manual control through the computer system.

Another view of MLS service is that it is sufficiently important that it should execute in a highly hardware-redundant computer such as a SIFT[12] or FTMP[13], in which the failure

probability is extremely low. However, although MLS processing could be performed on a highly reliable computer and is not essential following failure, for the purposes of this experiment, we have assumed that it is an example of a function that might be lost and has to be recovered. Were it operating in a weight or power restricted environment, such as a military aircraft or a spacecraft, it would not be possible to use extensive hardware replication to ensure safe operation of such functions. The MLS system represents a realistic example of the *type* of function that must be recovered in certain practical environments.

Although algorithmically the MLS system is complex, its overall structure is quite simple. Data is obtained from microwave receivers on the aircraft and used to compute position. Data acquisition requires confidence that a reliable source of MLS signals exists, as opposed to some form of radio noise. Thus the incoming data is checked over many real-time frames to ensure validity before any use is made of it. The data stream is filtered and various coordinate transformations are applied. The actual position, velocity and acceleration computations use a third-order complementary filter that depends on historic data for the various filter parameters. The MLS computations are performed every fifty milliseconds.

To recover MLS service if the FM/FC computer is lost, the alternate software must provide virtually identical computations and so the alternate software can be derived easily from the primary software. Fortunately, the MLS system can provide acceptable service if executed at half the normal rate and so the replacement software need be executed only every hundred milliseconds.

In order to start operating quickly, the replacement software must have available the status information about the incoming microwave signals and the historic information used as parameters in the filter. If this data is not consistent or not available, it can be recomputed in exactly the same way that the MLS software computes the information when it is initialized. The difficulty is that this computation takes several seconds since, for example, the determination that

the signals are reliable requires that they be monitored for many frames. To ensure that MLS service can be resumed immediately, the primary software must transmit its complement of data to the second processor on every real-time frame. For the MLS application, this amounts to a total of only approximately twenty real quantities.

Evaluation

Using the software structure just described for this application, most aspects of flexibility at development time are reduced very little by the application of non-transparent continuation. The reason is that the primary and alternate software components are mostly separate. Consider, for example, the Displays computer. It contains two systems that are almost completely isolated from each other, one for normal operation and one for operation after failure of the FM/FC computer. Since there are basically two systems, each can be dealt with separately and changes to either has little or no effect on the other. Of more importance is the impact that software changes on one processor have on the software of the other processor. However, again considering the Displays computer, changing the software used during normal operation has no impact on the software of the FM/FC computer. Clearly, changing the software that deals with failure of the FM/FC computer has no impact on the FM/FC computer software. This relatvely slight impact on flexibility is certainly not typical and probably will not be found in the analysis of other architectures.

An area where flexibility may be affected considerably is the addition of a processor. The addition of a processor to a dual-processor system is a major change and will certainly require extensive software modifications. Our analysis of this element of flexibility is not yet complete.

Recall that the major issue at execution-time is overhead. In this implementation, the overhead is different on the two computers. Both have their memory requirements approximately doubled. The alternate software plus the space needed to store copies of important data is

roughly the same size as the original software.

Processing overhead on the Displays computer is minimal. The alternate software to deal with failure of the FM/FC computer adds nothing since that software is normally suspended. Similarly, the reconfiguration task adds nothing during normal operation. The overhead used in supporting the failure semantics is also slight since it is a function of the amount of inter-task communication between machines. In examining this communication for this application, we see that the rate is relatively low. Several rendezvous take place during each fifty millisecond frame but this frame rate is not great.

The processing overhead on the FM/FC is also minimal. The alternate software to deal with failure of the Displays computer adds very little since it is not executed during normal operation although the selection of which software is executed must be made on each cycle and so the overhead of a conditional operation per cycle is incurred. Similarly, the reconfiguration task adds nothing during normal operation. The overhead used in supporting the failure semantics is slight since it is basically the same as incurred by the Displays computer.

Providing consistent data is an area where overhead will vary widely with the application. It is quite possible that algorithms will require large amounts of data in order to operate correctly. In this application, however, this is not the case. Considering the MLS software as an example, the filter parameters and the MLS status information must be made consistent across machines on each frame. The total volume turns out to be less than one hundred bytes, however, and this imposes virtually no burden on processing time or the data communications bus. This order of data volume is typical of other functions in this application also.

SUMMARY AND CONCLUSIONS

In this paper, we have described an experiment that we are conducting in an attempt to evaluate the non-transparent approach to recovery in distributed systems programmed in Ada that must tolerate processor failure. Ada does not address the issues raised by processor failure in distributed systems and the analysis that we are performing is based on a version of Ada in which no syntactic changes have been made but necessary extensions to the semantics have been added.

The experiment involves analysis of a typical application consisting of the software for the flight control system in an experimental commercial air transport. Only the major control aspects of the software and the concurrent parts are being examined. The sequential computations are being ignored, except for their general characteristics such as execution time and code volume.

It is not possible to draw a general conclusion about the utility of non-transparent continuation from a single experiment such as this. Also, the experiment described here is ongoing. There are numerous criteria that have to be evaluated under a variety of conditions. We plan to examine only a subset of the conditions, and only part of that analysis is complete.

At this point in the experiment we are very encouraged by the analysis and feel that non-transparent continuation is useful for this application at least. Programs that incorporate non-transparent continuation are not as flexible during development as those that ignore continuation or rely on transparent continuation. However, we remain convinced that some form of continuation is essential in crucial distributed systems and that transparent continuation is impractical. Our preliminary analysis also indicates that the overhead experienced by this application in support of non-transparent continuation is not excessive.

ACKNOWLEDGEMENTS

It is a pleasure to acknowledge J.R. Williams, E.H. Senn and R.M. Hueschen of NASA Langley Research Center, and W.C. Clinedinst and D.A. Wolverton of Computer Sciences Corportaion for many helpful technical discussions about the design and implementation of the ATOPS system. This work was supported in part by NASA grant NAG1-260.

REFERENCES

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

(2) Department Of Defense Requirements For High-Order Computer Programming Languages - STEELMAN, U.S. Department of Defense, 1978.

(3) 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*, to appear (also available as University of Virginia Department of Computer Science Technical Report No. TR-86-19).

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

(5) D. Cornhill, "Four Approaches To Partitioning Ada Programs For Execution On Distributed Targets", *Proceedings of the 1984 IEEE Computer Society 1984 Conference on Ada Applications and Environments*, St. Paul, Minnesota, October 1984.

(6) N.G. Leveson and P.R. Harvey, "Analyzing Software Safety", *IEEE Transactions On Software Engineering*, Vol. SE-9, pp. 569-579, September 1983.

(7) J.C. Knight 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.

(8) P.A. Alsberg and J.D. Day, "A Principle For Resilient Sharing Of Distributed Resources", *Proceedings Of The International Conference On Software Engineering*, San Francisco, October 1976.

(9) J.N. Gray, "Notes On Database Operating Systems", in *Operating Systems: An Advanced Course*, Springer-Verlag, New York 1978.

(10) HAL/S Language Reference Manual, Intermetrics Corporation, Cambridge, MA.

(11) M.E. Rouleau, "Analysis Of Ada For A Crucial Distributed Application", M.S. Thesis, Department of Computer Science, University of Virginia, 1987.

(12) J.H. Wensley, et al, "SIFT, The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE*, Vol. 66, pp. 1240-1254, October 1978.

(13) A.L. Hopkins, et al, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor For Aircraft", *Proceedings of the IEEE*, Vol. 66, pp. 1221-1239, October 1978.