

5)

**Estimating Task Execution Delay in a Real-Time System
via Static Source Code Analysis**

by

Steven B. Treadwell

Submitted to the
Department of Aeronautics and Astronautics
in Partial Fulfillment of the Requirements for the Degree of

**Master of Science
in Aeronautics and Astronautics**

at the

Massachusetts Institute of Technology

June 1993

© Steven B. Treadwell, 1993.

Signature of Author _____
Department of Aeronautics and Astronautics
May 7, 1993

Certified by _____
Professor Stephen A. Ward
Thesis Supervisor
Department of Electrical Engineering and Computer Science

Certified by _____
Dr. Richard E. Harper
Technical Staff, Charles Stark Draper Laboratory

Accepted by _____
Professor Harold Y. Wachman
Chairman, Department Graduate Committee
Aero

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUN 08 1993

Estimating Task Execution Delay in a Real-Time System

via Static Source Code Analysis

by

Steven B. Treadwell

Submitted to the Department of Aeronautics and Astronautics
in Partial Fulfillment of the Requirements for the Degree of

Master of Science

Abstract

In a hard-real-time system, it is critical that application tasks complete their iterative execution cycles within their allotted time frames. For a highly configurable parallel processing system, there exists an overwhelming set of hardware and software configurations, and it is useful to know a priori if a particular configuration satisfies hard-real-time constraints. This thesis presents an automated timing analysis tool which attempts to accurately characterize the timing behavior of the C3 Fault-Tolerant Parallel Processor (FTPP) developed at the Charles Stark Draper Laboratory. For each application task hosted by the FTPP, this automated tool performs a static source code analysis in an effort to estimate a lower bound on worst case execution delay. Then, using the specified mapping between software tasks and hardware processing sites, the analysis tool integrates the results of the individual task analyses in an effort to account for delays due to operating system overhead. The final portion of the analysis involves a prediction of possible performance failures based upon the given system configuration and the timing deadlines imposed by the FTPP's rate group scheduling paradigm. It is intended that the results of this timing analysis will help the user to develop a system configuration that optimizes throughput while minimizing the risk of performance failures.

Thesis Supervisor: Stephen A. Ward
Title: Professor of Electrical Engineering and Computer Science


Acknowledgments

I sincerely appreciate the support from everyone in the Advanced Computer Architectures group at CSDL. In particular, I would like to thank Rick Harper, Bryan Butler, and Carol Babikyan for their guidance, help and encouragement. I would also like to thank Bob, Chris, Anne, and Nate for making my MIT experience an enjoyable one.

This work was done at the Charles Stark Draper Laboratory under NASA contract NAS1-18565.

Publication of this report does not constitute approval by the Draper Laboratory of the findings or conclusions herein. It is published for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to the Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.



Steven B. Treadwell

Charles Stark Draper Laboratory hereby grants permission to the Massachusetts Institute of Technology to reproduce and to distribute this thesis in whole or in part.

Table of Contents

1. Introduction.....	9
1.1 Problem Statement.....	9
1.2 Objective.....	10
1.3 Approach.....	10
2. Background.....	13
2.1 Fundamentals of Fault Tolerance.....	13
2.2 Fundamentals of Byzantine Resilience.....	13
2.3 Illustrations of Byzantine Resilience.....	15
2.4 AFTA Hardware Architecture.....	19
2.5 Rate Group Scheduling.....	22
3. Preliminary Processing.....	27
3.1 Requirements.....	27
3.2 Inputs.....	28
3.3 Tools.....	29
3.4 Flow of Execution.....	30
3.5 The DCL Program.....	31
4. Software Analysis.....	33
4.1 Assumptions and Limitations.....	33
4.2 The Ada Software Structure.....	35
4.2.1 Subprograms and Packages.....	35
4.2.2 Tasks.....	36
4.3 Requirements of the Programmer.....	37
4.3.1 Comment Information.....	37
4.3.2 Naming Conventions.....	41
4.3.3 Undesirable Constructs.....	42
5. Abstraction.....	45
5.1 The Abstraction Methodology.....	45
5.2 The Motivation for Abstraction.....	46
5.3 Code Model Elements.....	47
5.4 Bottom-Up Construction.....	50
5.5 Expandability.....	52

6. Source Code Processing.....	53
6.1 The Big Picture.....	53
6.2 Establishing the Hierarchy.....	53
6.3 Code Modeling Tools.....	55
6.3.1 parse.....	55
6.3.2 read_list.....	56
6.3.3 get_line.....	56
6.3.4 search.....	58
7. Model Analysis.....	71
7.1 Model Preparation.....	72
7.2 Model Reduction.....	75
7.3 Execution Path Generation.....	80
7.4 Managing Model Analysis.....	87
8. Hardware Model Analysis.....	89
8.1 Introduction.....	89
8.2 Benchmarking.....	89
8.3 Path Comparison Calculation.....	95
8.4 Organizing Application Tasks.....	96
8.5 Predicting Performance Failures.....	98
8.6 Assumptions.....	102
8.7 Final Output File.....	103
9. Conclusions/Recommendations.....	105
9.1 Conclusions.....	105
9.2 Recommendations for Further Study.....	107
Appendix A HEADER.H Source Code.....	109
Appendix B START.C Source Code.....	113
main.....	113
read_list.....	114
get_line.....	115
search.....	116
write_file.....	117
extract_name.....	118

	strip.....	119
	get_arg.....	119
Appendix C	DCL Source Code.....	121
	ANALYZE.COM.....	121
	FIND.COM.....	121
Appendix D	FINISH.C Source Code.....	123
	main.....	123
	match_up.....	124
	process_list.....	126
	task_parse.....	128
	find_packages.....	130
	update_pkg_list.....	131
	parse.....	132
	check_overrun.....	134
	read_list.....	138
	get_line.....	140
	search.....	142
	parse_comment.....	147
	write_file.....	148
	with_found.....	151
	pkg_found.....	152
	task_found.....	153
	packetize.....	153
	proc_found.....	154
	find_parameter.....	155
	process_loop.....	157
	for_loop_found.....	158
	end_found.....	159
	valid_call.....	162
	print_line.....	162
	target_found.....	163
	eval_range_num.....	164
	eval_natural_num.....	165
	eval_complex_num.....	166
	eval_simple_num.....	166
	evaluate.....	167

	<code>print_procedures</code>	168
	<code>find_worst_path</code>	169
	<code>reduce_model</code>	170
	<code>nest_level</code>	171
	<code>match_loops</code>	173
	<code>crunch</code>	175
	<code>check_ctrs</code>	177
	<code>generate_paths</code>	178
	<code>decide</code>	180
	<code>parameterize</code>	182
	<code>model_ok</code>	184
	<code>calculate_time</code>	186
Appendix E	External Files	187
	<code>key_words.dat</code>	187
	<code>constants.dat</code>	187
Appendix F	An Illustrative Example	189
	<code>task_list.ada</code>	190
	<code>app_test.ada</code>	194
	<code>sys_fdi.ada</code>	199
	<code>test_code.ada</code>	206
	<code>first_package.ada</code>	206
	<code>second_package.ada</code>	207
	<code>task_names.dat</code>	208
	<code>list_of_tasks.dat</code>	208
	<code>filenames.dat</code>	208
	<code>results.dat</code>	209
	<code>errors.dat</code>	211
Appendix G	A Checklist for Adding Critical Constructs	217
Appendix H	References	219

List of Figures

Figure 2-1,	Minimal 1-Byzantine Resilient System.....	15
Figure 2-2,	1-Round Data Exchange.....	16
Figure 2-3,	All FCRs Agree.....	17
Figure 2-4,	First Round Exchange.....	18
Figure 2-5,	Second Round Exchange.....	18
Figure 2-6,	Results from Example #2.....	19
Figure 2-7,	AFTA Hardware Configuration.....	20
Figure 2-8,	AFTA's Virtual Bus Topology.....	21
Figure 2-9,	Rate Group Frame Organization.....	23
Figure 2-10,	Scheduling for a Single Rate Group Frame.....	24
Figure 3-1,	Sample Task Specification File Entry.....	28
Figure 3-2,	Task Information Storage Format.....	30
Figure 3-3,	Preliminary Processing File Flow.....	31
Figure 4-1,	Possible Distribution of Execution Times.....	34
Figure 4-2,	Sample Rate Group Task.....	36
Figure 4-3,	Variable Tracing Example #1.....	38
Figure 4-4,	Variable Tracing Example #2.....	39
Figure 4-5,	Examples of Comment Information.....	40
Figure 4-6,	Potential Infinite Loop.....	43
Figure 5-1,	List of Critical Constructs.....	46
Figure 5-2,	Sample Source Code Segment.....	49
Figure 5-3,	Sample Source Code Model.....	49
Figure 5-4,	An Ideal Software Hierarchy.....	51
Figure 6-1,	Timing Analysis Hierarchy.....	54
Figure 6-2,	The parse Algorithm.....	57
Figure 6-3,	The search Algorithm.....	59
Figure 6-4,	Ada's Framed Constructs.....	61
Figure 6-5,	Example of end Statement Ambiguity.....	62
Figure 7-1,	A Sample if Construct.....	73
Figure 7-2,	A Sample Nested if Construct.....	74
Figure 7-3,	A Sample Nested Loop and its Model.....	76
Figure 7-4,	Format for a COUNTER_SET Entry.....	77
Figure 7-5,	An Updated Model.....	77

Figure 7-6,	A Nested <code>if</code> Construct.....	80
Figure 7-7,	A Nested <code>if</code> Decision Tree.....	81
Figure 7-8,	The <code>generate_paths</code> Algorithm.....	85
Figure 8-1,	Minor Frame Overhead Model.....	90
Figure 8-2,	Rate Group Frame Organization	98
Figure 8-3,	A Simplified View of Rate Group Scheduling.....	101

Chapter 1

Introduction

1.1 Problem Statement

The Fault-Tolerant Parallel Processor (FTPP) developed at the Charles Stark Draper Laboratory is a computer architecture aimed at achieving high throughput while maintaining a high level of reliability. These are necessary qualities for a computing system that could be called upon to perform flight-critical and mission-critical tasks such as those found in an aircraft flight control system. The FTPP utilizes multiple processing elements (PEs) operating in parallel to achieve high throughput, and it maintains high reliability through implementation of PE redundancy and Byzantine resilience. The high throughput of the FTPP makes it an ideal host for hard-real-time applications, and its custom-built operating system uses a rate group scheduling paradigm to properly schedule iterative execution of real-time tasks. Application tasks are divided into rate groups according to their required frequency of execution, and the operating system schedules individual tasks for execution at regular, predefined intervals. For a real-time system, it is critical that each task complete its execution cycle within its allotted time frame; an inability to complete execution on time results in a condition known as a performance failure.

The current incarnation of the FTPP is the Army Fault Tolerant Architecture (AFTA). It is designed to be highly configurable and thus capable of supporting a varying set of mission requirements. The AFTA can support as many as forty individual processing elements, and these are organized into a flexible set of virtual groups (VGs) to achieve PE redundancy. For any particular mission, the AFTA is uploaded with a suite of application tasks, each of which may execute on one or more virtual processing groups. Individual tasks vary according to function, required level of redundancy, required frequency of execution, and expected execution delay. Given the variability of the AFTA hardware and software configuration, one may produce an overwhelming set of all possible mappings between tasks and processing sites. For a hard-real-time system, it is critical for the task workload to be properly distributed among the virtual groups so that all tasks are able to complete their iterative execution cycles within their allotted time frames. In order to identify such a task distribution, one may use a form of operational trial and error, but it is certainly preferable to know in advance if a chosen configuration of hardware and software satisfies necessary real-time constraints. For that purpose, this thesis presents an automated software tool to perform a timing analysis of any given system configuration.

1.2 Objective

The automated timing analysis takes into account the full system configuration--both hardware and software. It performs a static analysis of the source code for each application task and uses system performance data to estimate a least upper bound on task execution time. The timing analysis relies heavily on source code modeling techniques, and the limitations of modeling prevent a precise calculation of execution time. A worst case scenario is considered in the analysis, and a minimum (rather than maximum) worst case delay is defined using the model. Once a least upper bound is established for every task, the tasks are categorized according to their virtual group and rate group specifications, and comprehensive calculations are made for each virtual group to determine if the overall system can satisfy real-time constraints under worst case conditions. This calculation is known as the frame overrun check, and it takes into account the following: hardware configuration, application task characterization, task scheduling overhead, and operating system performance data.

The use of the analysis tool is not valuable solely for the overrun prediction; rather, the overrun check is simply the most comprehensive result produced. The more important results are the intermediate values used in performing the overrun check. These include the delay of the rate group dispatcher and the parameterizations of the individual application tasks. One of the major goals of this analysis is to properly characterize the software tasks for timing estimation, code optimization, and for further analysis of global message traffic and virtual group phasing (to be accomplished by other tools). After a single configuration analysis, it should be readily apparent what types of changes could be made to the system for better performance results. These changes might include streamlining application task code, altering the mapping between tasks and virtual groups, adjusting the virtual group configuration of the AFTA hardware, or switching the rate group specification of one or more application tasks.

1.3 Approach

This timing analysis uses a modeling approach to account for the combined behavior of the hardware and software in any given system configuration. The analysis tool examines the Ada source code for each application task and develops a model for its flow of execution. From this model, every possible path of execution is generated and characterized according to a predefined set of parameters. Using performance data for the AFTA operating system, the analysis tool compares the estimated execution times of all paths and thus identifies the worst case path. Once a worst case path is defined for all application tasks, this data is input to a model of the hardware configuration. The

hardware model primarily accounts for the mapping between software tasks and processing elements, and it uses the worst case path parameterizations to determine if the full system can satisfy real-time constraints under worst case conditions.

All the analysis functions described above are performed by a combination of two programs written in C, which require minimal user interaction. Also, two “.com” files written in Digital Command Language (DCL) are used for file searching operations and proper execution sequencing of the two C programs. The results produced by the analysis are stored in two machine-readable files; one contains the numerical results and the other serves as an error log.

Chapter 2 Background

2.1 Fundamentals of Fault Tolerance

A computing system designated to perform mission-critical and flight-critical tasks must maintain a high level of reliability since faulty operation could cause loss of aircraft control or at least compromise mission effectiveness. The total reliability of a system is a function of the individual reliabilities of its components and their working relationships with one another. The reliability of individual components is always bounded and can usually be determined through experimentation; the goal of fault tolerance is to use strategic component redundancy to achieve a system reliability which is greater than that of the individual components. By definition, a fault-tolerant system must be able to survive erroneous operation by some subset of its components and still properly perform all assigned tasks [HAR91].

A typical reliability goal for a flight-critical computer system is 1 failure in 10^9 hours, while the components of that system may exhibit failure rates on the order of 1 in 10^4 hours [HAR91]. Some sort of redundancy scheme must be utilized to build a system that is 10^5 times more reliable than its individual components. One approach is to first examine all possible failure modes, the extent of their effects, and their associated probabilities of occurrence. Then the system is designed to protect against all potentially fatal failure modes which are judged to have a significant probability of occurring, and the design must address a sufficient number of probable failure modes such that the system reliability goal is achieved. This method is not only cumbersome and inexact, but it is also difficult, if not impossible, to validate the reliability of the design. A mathematical validation of the design's reliability requires that for every error which occurs, the probability that the design does not adequately protect against that error must be less than 10^{-5} [HAR91]. It is certainly conceivable that system designers could overlook significant types of erroneous behavior that would eventually surface in field operation at the expense of system reliability. It is therefore preferable to employ a design methodology that addresses only the number of potentially fatal component failures and ignores the exact behavior of faulty components; this is the objective of the Byzantine resilience approach to fault tolerance.

2.2 Fundamentals of Byzantine Resilience

Byzantine resilience guarantees proper operation of a system for a predefined number of component failures, regardless of the specific nature of the individual failures.

The concept of Byzantine resilience is derived from the solution to the Byzantine Generals Problem; it is stated as follows:

1. Imagine several divisions of the Byzantine army camped around an enemy stronghold; each division has its own commanding general.
2. Upon observation of the enemy, the generals must decide whether to attack or retreat. They communicate with one another only by messenger.
3. Some generals may be traitors and thus try to prevent the loyal generals from reaching an agreement. All messengers are considered loyal; traitorous activity by a messenger is treated as traitorous activity by the general sending the message.
4. The objective is to develop an algorithm to guarantee that all loyal generals follow the same plan of action, and no small number of traitors can cause the loyal generals to adopt a bad plan [LAM82].

This problem is analogous to that of designing reliable computer systems. The commanding generals represent processors in a redundant configuration, the traitors represent faulty processors, and the messengers correspond to the interprocessor communication links [HAR91]. Using this analogy, the problem may be restated as follows:

1. A redundant computer system consists of multiple processors.
2. The processors utilize identical inputs to produce required results. They communicate with each other over data links.
3. Some processors may be faulty and may demonstrate malicious and even intelligent behavior. Faulty communication links can be analytically treated as faulty processors.
4. The objective is to force the system outputs to reflect an agreement among the set of non-faulty processors and to effectively mask the behavior of faulty processors.

The solution to this problem is best understood after explaining some terminology. The physical components of a Byzantine resilient computer system are typically organized into a number of subsystems referred to as Fault Containment Regions (FCRs). Each FCR has a certain level of processing power and maintains communication links with other FCRs in the system. By definition, any fault which occurs within an FCR should not be propagated outside that subsystem to other FCRs. A system is said to be f -Byzantine resilient if it can withstand a number of failures that is less than or equal to f . One should note from the statement of the problem that an FCR failure can denote any type of malicious or even intelligent behavior, and this ensures

proper coverage of all possible failure modes as long as the number of failures is less than or equal to f . The solution to the Byzantine Generals Problem can be transformed into a set of implementation requirements for an f -Byzantine resilient system; these are summarized as follows:

1. There must be at least $3f+1$ FCRs [LAM82].
2. Each FCR must be connected to at least $2f+1$ other FCRs through disjoint communication links [DOL82].
3. For information emanating from a single source, there must be at least $f+1$ rounds of communication among FCRs [FIS82]
4. The activity of the FCRs must be synchronized to within a known and bounded skew [DOL84].

For a 1-Byzantine resilient system, there must be four FCRs with each one uniquely connected to the other three; and for single sourcing of information, two rounds of communication are required. A minimal 1-Byzantine Resilient System is shown in Figure 2-1.

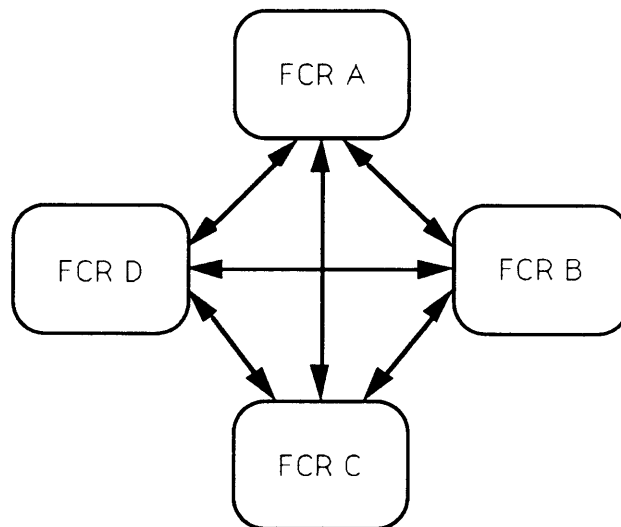


Figure 2-1, Minimal 1-Byzantine Resilient System

2.3 Illustrations of Byzantine Resilience

The following two examples illustrate the Byzantine resilience approach to fault-tolerant computing.

The first example shows how communication among four FCRs in a 1-Byzantine resilient configuration can overcome faulty operation by a single FCR. Suppose each FCR contains a single processor and all four processors perform the same operation. Also assume that FCR A is responsible for conveying the system outputs to an external

actuator. Simultaneously all four processors produce results for some required computation, and the system must send these results to the actuator.

Figure 2-2 shows that the processors in FCRs B, C, and D all reach a result of '1' while the processor in FCR A makes an error and submits a '0' as its result. In order to determine the output for the system, the FCRs perform a single round of communication as indicated by the arrows, and each FCR then knows what results were reached by all the other FCRs. A majority vote of the four sets of data is performed by each FCR, and since every subsystem works with the same set of four data values, the FCRs necessarily agree upon the proper output for the system. Figure 2-2 shows that each FCR has a set of three '1's and one '0' upon which to vote; thus they must reach the same conclusion.

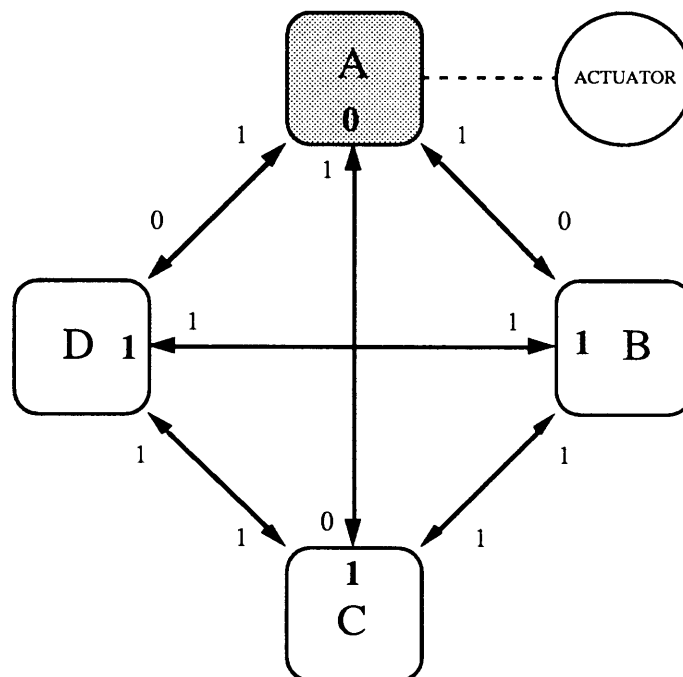


Figure 2-2, 1-Round Data Exchange

The result of the inter-FCR communication and voting is that the processor in FCR A is outvoted and the system result is given as a '1.' This example assumes that FCR A itself is not faulty; rather, the processor *in* FCR A experiences a *temporary* malfunction. Despite this malfunction, the actuator associated with FCR A still receives the correct system output, and this is shown in Figure 2-3. Thus, the temporarily faulty operation of a single processor is masked in the system output of this 1-Byzantine resilient configuration.

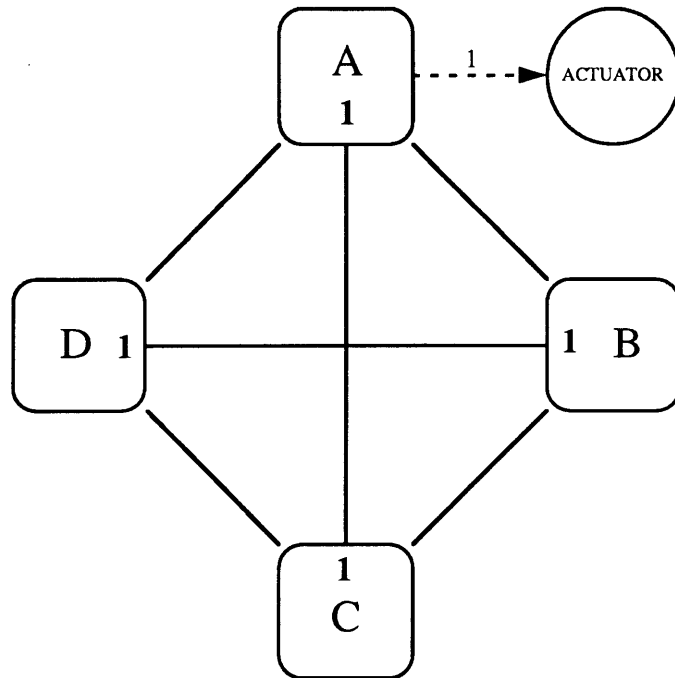


Figure 2-3, All FCRs Agree

The example above illustrated the use of the “1-round” data exchange. This type of communication is known as a voted message, and it is used when an exact consensus is expected among redundant processors performing the same function. Another necessary form of communication requires two rounds of data exchange, and this is known as passing a source congruency message. A “2-round” exchange is required when a single data source sends information to a specific processor or a group of processors [AFTA91]. The next example illustrates this type of communication for a 1-Byzantine resilient system.

Suppose there is a sensor associated with FCR A, and it wishes to send data to processors in each of the four FCRs. Let this data be represented by a binary value of ‘1.’ Figure 2-4 shows the sensor sending its data to FCR A; FCR A then transmits this information to all the other FCRs. This is the first round of data exchange. Note that the data properly reaches FCRs C and D, but an error between FCRs A and B causes B to read a value of ‘0.’ This type of fault is equivalent to traitorous activity by FCR B or FCR A; without loss of generality, it is assumed that FCR B is faulty.

The first round of communication is followed by a second round in which the FCRs exchange the values they received in the first round. Thus FCRs A, C, and D send out ‘1’s to all their neighbors, and the traitor, FCR B, sends out ‘0’s to all of its neighbors. Assume for now that the failure of FCR B to properly read A’s first message

was a transient fault and does not occur on the second round. Figure 2-5 shows this second round activity. At this point, all four FCRs have been presented with an identical set of four values, and they vote individually to reach the same result.

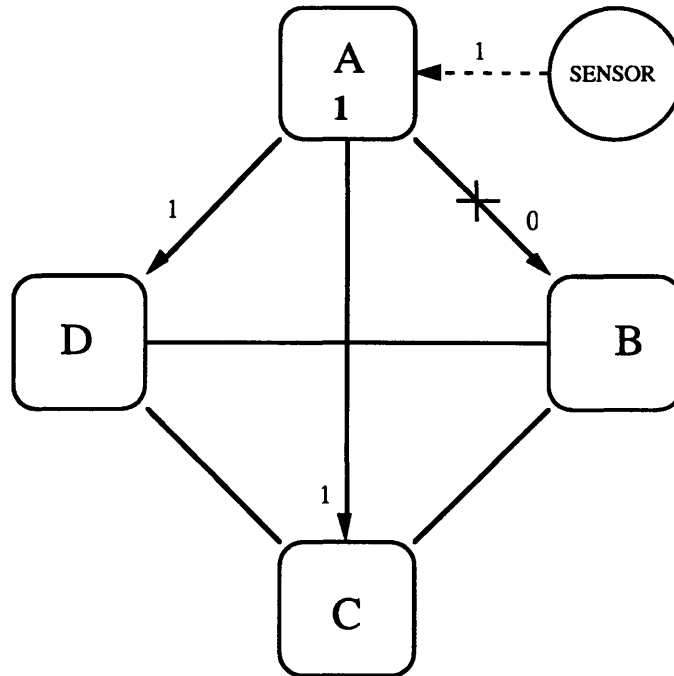


Figure 2-4, First Round Exchange

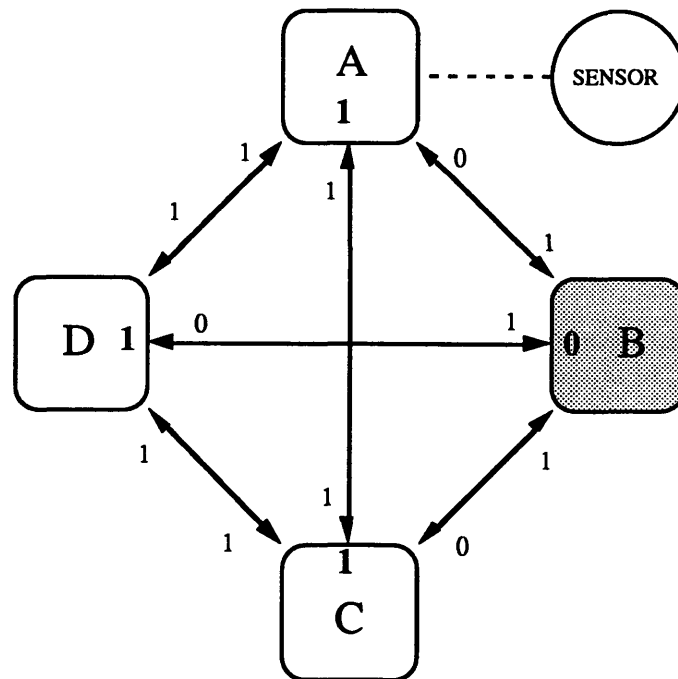


Figure 2-5, Second Round Exchange

Figure 2-6 shows that all four FCRs reach a consensus value of '1' and pass this value on to the processors associated with each FCR. Thus the second round of exchanges allows the transient fault on FCR B to be overcome so that the processor associated with FCR B could receive the proper value from the sensor attached to FCR A.

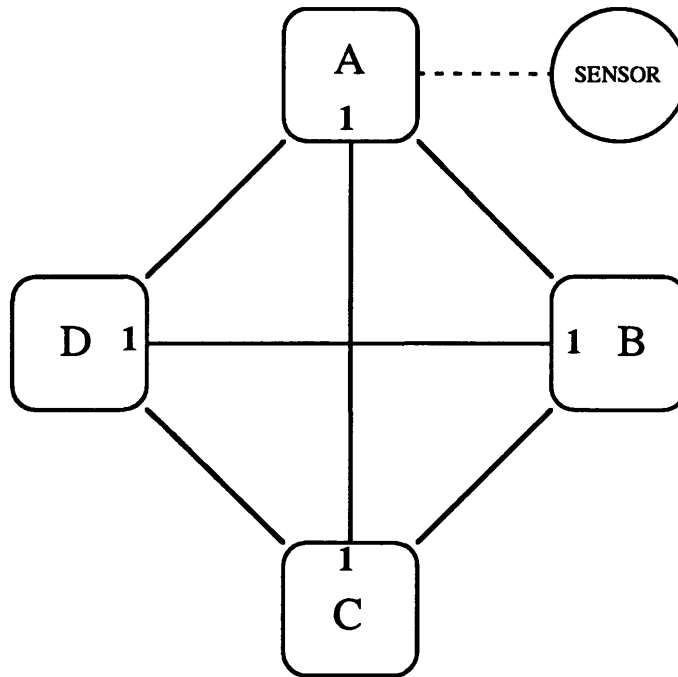


Figure 2-6, Results from Example #2

Now consider the case where FCR B exhibits permanent faulty behavior rather than the transient fault described previously. In this case, B could send out any random value to its neighbors and also read back random values for the messages received from its neighbors. This means it would be improper to assume that FCR B votes upon the same set of data as the other FCRs; it must be assumed that FCR B reaches the wrong value and its processor therefore receives faulty data from the sensor attached to A. This scenario still does not compromise the effectiveness of the system as a whole, for the perceived faulty operation of FCR B and its processors is masked by the proper operation of the remaining FCRs. This property is demonstrated in the first example, where faulty operation in A is masked by correct operation of FCRs B, C, and D.

2.4 AFTA Hardware Architecture

The Army Fault Tolerant Architecture is designed as a 1-Byzantine resilient system organized as a cluster of either four or five fault containment regions. Each FCR consists of a network element (NE), 0 to 8 processing elements (PEs), and 0 or more

input/output controllers (IOCs) for interfacing with external devices. Figure 2-7 illustrates the hardware configuration of AFTA.

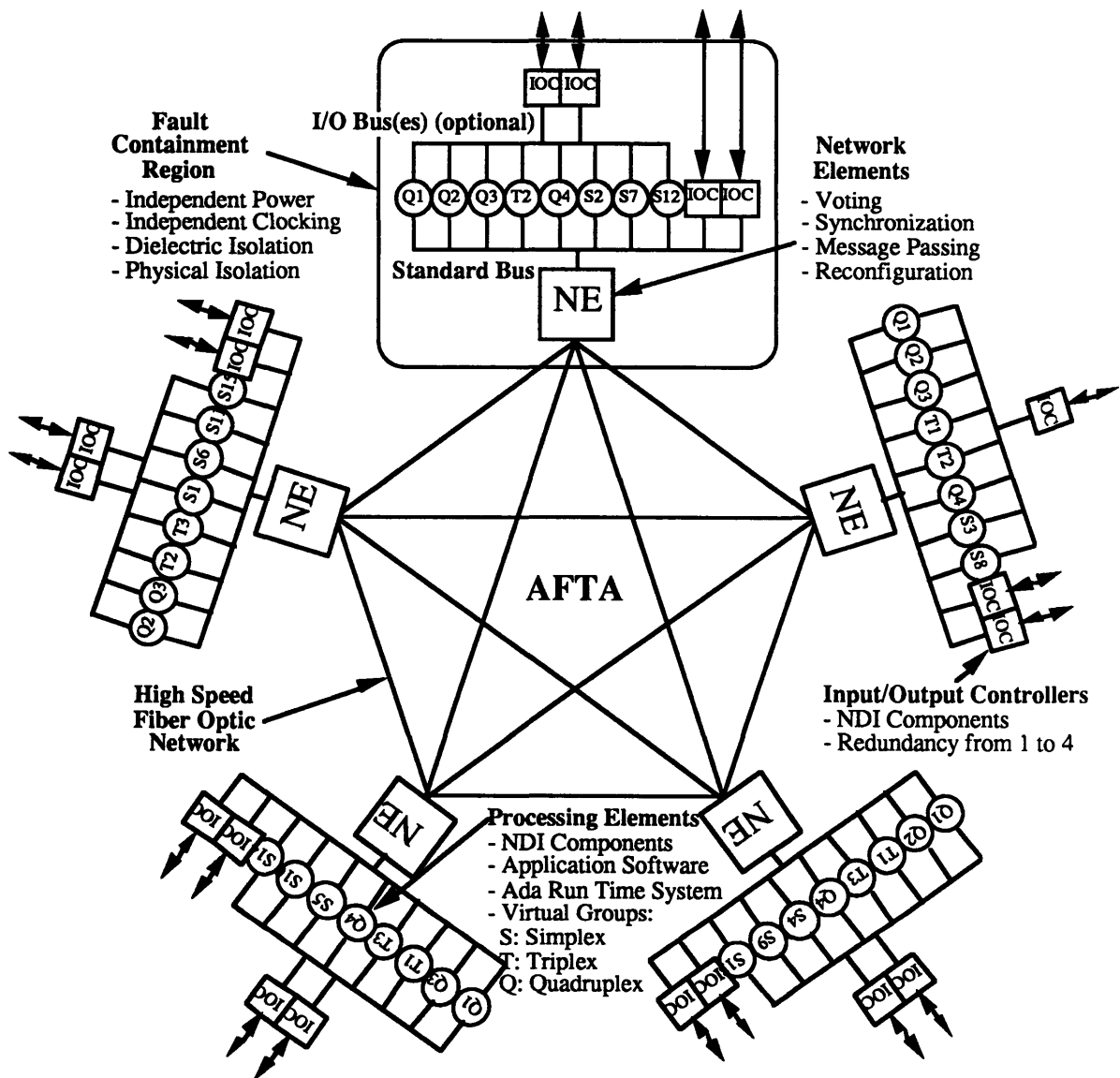


Figure 2-7, AFTA Hardware Configuration [AFTA91]

Byzantine resilience requires that faults within one FCR do not alter the operation of another FCR; thus the AFTA design allows for both physical and dielectric isolation of FCRs. Every FCR maintains independent clocking and has its own power source, backplane, and chassis. The only physical connection between FCRs is a fully connected high speed fiber optic network which provides reliable communication without compromising the dielectric isolation [AFTA91].

The processing elements are standard processor boards with local memory and miscellaneous support devices; laboratory prototype tests use Motorola 68030 VME processor boards. The PEs are organized into virtual groups (VGs) with one, three, or four processors per group, and these groups are referred to as simplex, triplex, and quadruplex, respectively. VGs with multiple processors execute an identical suite of tasks on each PE to provide the redundancy needed for fault tolerance, and Byzantine resilience requires that every member of a triplex or quadruplex VG resides in a different FCR. Each VG operates independently of the others, and the combined processing power of multiple VGs functioning in parallel is what allows the AFTA to satisfy its high throughput requirement. A minimal AFTA configuration consists of four FCRs hosting a single virtual group of three PEs. A maximal configuration supports forty PEs divided evenly among five FCRs [CLAS92]. The virtual group configuration in this case can range from forty simplexes to ten quadruplexes. Note that the specific virtual group configuration for a given system setup depends upon performance, reliability, and availability constraints, and can include mixed redundancy virtual groups. Figure 2-7 illustrates the organization of forty PEs into their respective VGs as well as the physical separation of the individual members of redundant VGs.

The core of each FCR is the network element. The NE maintains the fiber optic links between the FCRs and also keeps the FCRs properly synchronized. The network element is designed to implement all message passing and data voting protocols required by Byzantine resilience, and it is the NE that actually carries out the communication between the PEs within a VG and between the VGs themselves. The PEs communicate with the network element over a standard bus such as the VME bus and the NEs talk to one another via the fiber optic network. The network element is responsible for receiving messages from the PEs in a 64-byte packet format, transmitting message packets over the fiber optic network, storing message packets destined for PEs within its FCR, and notifying its PEs of message packet arrivals. The AFTA operating system works closely with the NE hardware to ensure that the necessary communication protocols are implemented smoothly, and the result is that the entire AFTA communication network can be viewed as a virtual bus topology with all processors and virtual groups tied to the bus [AFTA91]. This simplified view of the AFTA is shown in Figure 2-8. Message passing between virtual groups occurs asynchronously over this virtual bus, and the hardware is designed to guarantee that message packets are delivered correctly and in order.

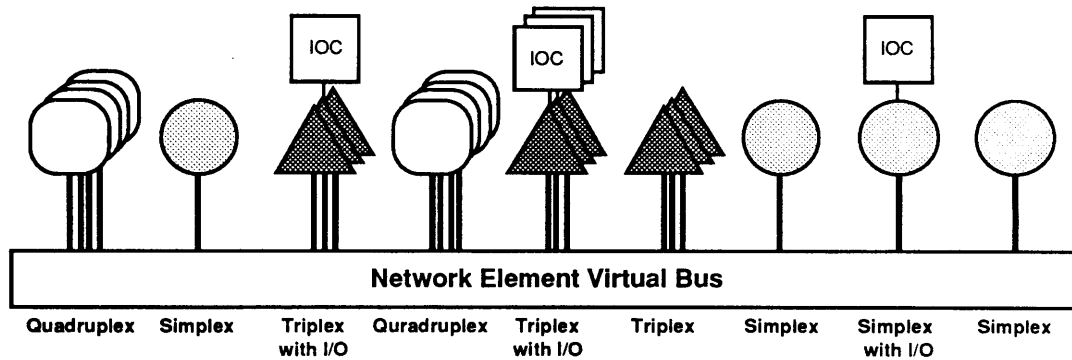


Figure 2-8, AFTA's Virtual Bus Topology [AFTA91]

The AFTA provides a unique combination of parallel processing and fault tolerance capabilities, and the inherent complexity of these combined disciplines could cause unnecessary difficulty for software developers. Fortunately, the custom design of the network element and the AFTA operating system make the system's hardware configuration relatively transparent to application task programmers [HAR91]. The programmer does not need to understand the requirements of Byzantine resilience or the subtleties of parallel processing. The operating system provides a set of message passing services that allow an applications developer to perform intertask communication via simple function calls. Thus the programmer must know only the global communication identification for the tasks with which he communicates; he may ignore the actual virtual group configuration and physical system setup. This thesis primarily views the AFTA at this level of abstraction.

2.5 Rate Group Scheduling Overview

The AFTA is designed specifically to support hard-real-time application tasks, and a rate group scheduling paradigm is utilized to achieve hard-real-time response for both periodic and aperiodic tasks [CLAS92]. A hard-real-time task is a process that is executed in an iterative manner such that every execution cycle is completed prior to a predefined deadline. Typically a task is allotted a certain time frame in which to execute, and if execution is incomplete at the time of the deadline, a frame overrun condition is in effect. A frame overrun denotes a failure on the part of system task scheduling, and it could lead to catastrophic results if the outputs of a task are critical to a function such as flight control.

The AFTA's processing resources and software task assignments are logically divided among the system's virtual groups. Each virtual group is responsible for scheduling the execution of its own set of tasks, and it utilizes a combination of two

scheduling algorithms. The first is rate group scheduling; it is useful for tasks with well-defined iteration rates and guaranteed maximum execution times (i.e. flight control functions). The second method is aperiodic non-real-time scheduling, and this is used for non-real-time tasks whose iteration rate is unknown or undefined (i.e. mission planning). Note that the task scheduling algorithms do not allow non rate group tasks to disturb the critical timing behavior of rate group tasks [CLAS92].

In the rate group scheduling paradigm, the real-time tasks on a single VG are categorized according to their required iteration rates. Presently the AFTA supports four rate group designations; the names, frequencies, and frame allocations of these groups are summarized in Table 2-1 [AFTA91].

Table 2-1, Rate Group Designations

Rate Group Name	Iteration Rate	Iteration Frame
RG4	100Hz	10ms
RG3	50Hz	20ms
RG2	25Hz	40ms
RG1	12.5Hz	80ms

At any given instant of time, all four rate group frames are simultaneously active, although the processing power of the virtual group is dedicated to only one task within one rate group. Rate group preemption is allowed such that tasks within a faster rate group are always able to interrupt tasks within a slower rate group and thus divert processing power to ensure that higher frequency tasks complete execution before their next deadline. For example, if an RG3 task is executing when a new RG4 frame begins, the RG3 task is suspended in favor of RG4 tasks. All RG4 tasks execute to completion, and then execution of the suspended RG3 task is resumed.

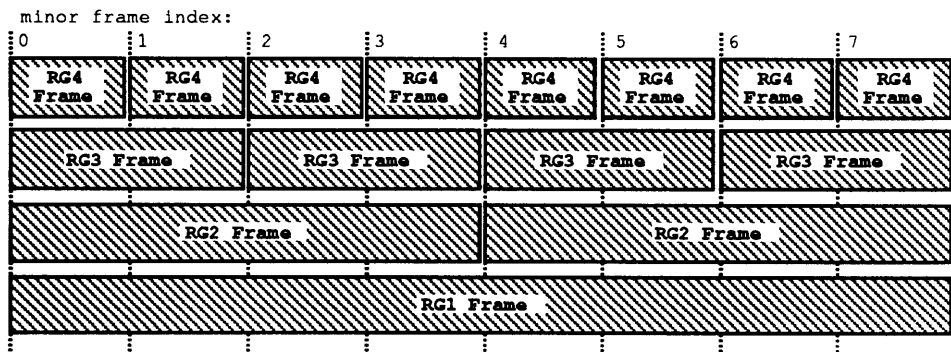


Figure 2-9, Rate Group Frame Organization

Figure 2-9 serves as a pictorial explanation of the organization of rate group frames relative to one another for a single virtual group. Notice that for an 80ms slice of time (one RG1 frame), RG4 tasks are executed 8 times, RG3 tasks are executed 4 times, RG2 tasks are executed twice, and RG1 tasks are executed once.

For a given rate group frame, the VG schedules tasks on a static, non preemptive basis [CLAS92]. In other words, tasks within the same rate group cannot interrupt one another even though higher frequency tasks are allowed to interrupt, and the ordering of tasks within the rate group frame depends upon a task priority assignment made during system initialization. Despite the preemptive activity between tasks of different rate groups, each task on a VG should eventually execute until it reaches a state of self-suspension. For each rate group frame, every task in that rate group must enter self-suspension before the frame boundary; failure to do so constitutes a frame overrun. When a task suspends its own execution, it has effectively completed its execution cycle and the VG's processing power is passed on to another task. At the beginning of a new rate group frame, every task in the rate group is once again scheduled for execution and resumes execution at the point where it last suspended itself. Figure 2-10 shows how tasks are scheduled within an arbitrary rate group frame. Notice that incoming messages are delivered to the rate group tasks at the beginning of the frame and outgoing messages are queued during the frame and then transmitted at the end of the frame.

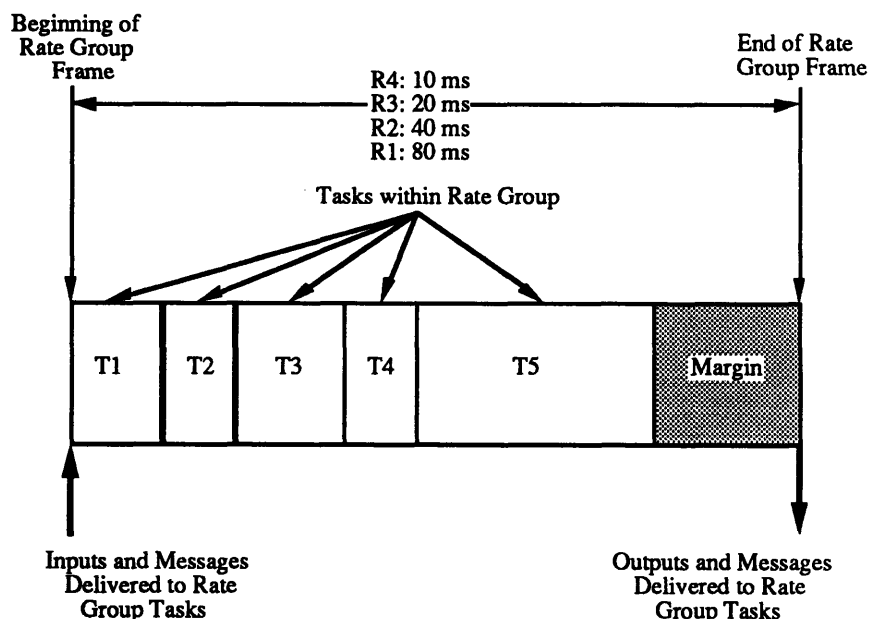


Figure 2-10, Scheduling for a Single Rate Group Frame

The actual task scheduling for a virtual group is performed by an RG4 task known as the rate group dispatcher. It executes at the beginning of every minor frame (RG4 frame) and schedules execution of all tasks belonging to rate groups whose frame boundaries coincide with the beginning of the current RG4 frame. Refer to Figure 2-9 for a good illustration of the frame boundary synchronization.

The rate group dispatcher also performs several bookkeeping functions for the operating system, and it triggers transmission of the messages that were queued during the rate group frames that were just terminated. Any task overruns from the previous frames are detected by the RG dispatcher, and the error handling system is notified. In the case of an overrun, the RG dispatcher detects which tasks were unable to complete execution and enter self-suspension [CLAS92]. It is possible that a task which causes an overrun actually completes its execution and effectively forces lower priority and lower frequency tasks to remain incomplete. Thus, the source of this type of overrun error is very difficult to trace, and it is intended that the type of a priori configuration timing analysis proposed by this thesis will prevent the occurrence of such errors.

Chapter 3

Preliminary Processing

3.1 Requirements

The AFTA timing analysis is divided into three distinct stages -- preliminary processing, software modeling, and hardware modeling. Each stage builds upon the results of the previous stage(s) and contributes to the formulation of final results. The analysis begins with the preliminary processing stage, and its primary function is to provide the other stages with an accurate and concise picture of the system configuration. One of the goals of the timing analysis is to minimize user interaction so that the computer bears the brunt of the analysis workload, and the preliminary processing phase is designed to gather its information in an automated fashion from existing software files and avoid querying the user about the system setup. The user is only required to provide the name of the current task specification file; the analysis software does the rest.

Both the hardware and software analysis stages depend upon the configuration information provided by the preliminary processing stage; however, these two types of analysis view the system from different perspectives. The software analysis develops models of the application task source code, and it sees the system as a collection of task instantiations. The hardware analysis performs calculations relevant to the rate group timing deadlines, and it sees the system as a collection of virtual groups. The goal of the preliminary processing is to find the information required by both the software and hardware analyses and to provide it in a format that is useful to both.

The gathering of system configuration data focuses upon the individual application tasks, and the preliminary processing stage seeks answers to the following questions:

1. What are the names of all application tasks in the suite?
2. On which virtual group(s) does each task reside?
3. To which rate group does each task belong?
4. What are the message passing limitations for each task?
5. Which file contains the source code for each task?

With regard to formatting this information, notice that the setup data is easily organized according to tasks. This is a convenient format for the software analysis, but it is also easily transformed into a virtual group format for the convenience of the hardware analysis. This transformation is explained in Chapter 8.

3.2 Inputs

Ada software development and maintenance for the AFTA currently takes place on a VAX minicomputer. During AFTA testing, application task code is transferred from the VAX file server to the AFTA's processing elements, and in the process, the operating system is provided with a task specification file, which essentially explains the software setup to the operating system. The preliminary processing phase utilizes this file to gather most of its configuration information. The file is organized as a series of records with each record containing information about a single task instantiation. A sample entry is shown below in Figure 3-1.

```
1 => (  
  gcid => gcids.fdi,  
  gtid => gtids.fdi,  
  location => config.all_vg,  
  vg => 0,  
  rg => config.rg4,  
  precedence => 12,  
  max_xmit_size => 200,  
  max_xmit_num => 10,  
  max_rcve_size => 200,  
  max_rcve_num => 20,  
  num_ions => 0,  
  ions => (  
    others => (  
      num_chains => 0,  
      chains => (  
        others => (E => false, D => false, C => false,  
                  B=>false, A => false))))))
```

Figure 3-1, Sample Task Specification File Entry

Since the format here is somewhat cryptic, it requires some explanation. Figure 3-1 shows the first task entry in the file, as denoted by the "1=>." The `gcid` and `gtid` lines refer to the task's global communication identification and global task identification, respectively, and these are necessary for message passing purposes. The name of the task is `fdi`, and it is extracted from the line "`gtid=>gtids.fdi`." `fdi` is a software service that performs fault detection and isolation; by convention, a "_t" is appended to the task name, and the source code for `fdi` is found in a task body named `fdi_t`. Note

that there is not necessarily any correlation between the name of the task and the name of the package or file where it is held. The `location` line shows that `fdi` is configured to execute on all virtual groups in the system; a task can be configured to run on all VGs simultaneously or only on one VG. The “`vg=>0`” line shows which VG hosts the task; in this case, “0” is used because the task is mapped to all operational VGs. The `rg` line indicates that `fdi` is an RG4 task, and its `precedence` value determines how it is mapped statically within an RG4 execution frame. The next group of four lines refers to the message buffering requirements of the task. `max_xmit_size` and `max_rcve_size` indicate the maximum size, in bytes, of messages that the task is allowed to transmit and receive. Likewise, `max_xmit_num` and `max_rcve_num` determine the maximum number of messages a task can send and receive during a single execution cycle. The final portion of the record concerns I/O requests, and this information is currently irrelevant to the timing analysis.

Similar to the AFTA application software, the timing analysis program is resident on the VAX. All file manipulations in the course of the analysis take place on the VAX, completely separate from the actual FTTP. The user provides the analysis program with the task specification filename, and the preliminary processing begins with an examination of this file.

3.3 Tools

The preliminary processing stage consists of one C program named `START.C` and one DCL program called `FIND.COM`. The execution of the C program is explained in this section and the following section; the DCL file is examined in Section 3.5. Source code listings are included as Appendices B and C, respectively.

`START.C` utilizes some simple tools to examine the task specification file and pass on the information it finds to the DCL program and to future stages of the timing analysis. The first tool is a procedure called `read_list`. Its function is to read a pre-defined file which lists a series of key words that are needed by `START.C` when examining the task specification list. `read_list` opens the file, grabs each key word individually, and stores it in a structure called `search_list`.

The next procedure is `get_line`; its function is to pull characters from the task specification file and assemble them into words and assemble the words into individual lines. In this case, a line refers to the string of characters found between carriage returns. Whenever it is called, `get_line` finds and returns the next line in the file. The line is stored as a collection of words in a structure called `this_line`; all white space and comments are deleted from the line.

The procedure called `search` is the workhorse of `start.c`. Its job is to search a single line of the task specification file looking for the key words stored in `search_list`. When a key word is found, it means that there is vital information in that line, and a specific procedure is called to extract that information. All relevant data found in the file is stored in a structure called `task`; its format is shown in the pseudocode of Figure 3-2:

<code>task record</code>	<code>(structure)</code>
<code>name</code>	<code>(string)</code>
<code>virtual group</code>	<code>(integer)</code>
<code>rate group</code>	<code>(integer)</code>
<code>message buffers</code>	<code>(structure)</code>
<code>max transmit size</code>	<code>(integer)</code>
<code>max transmit number</code>	<code>(integer)</code>
<code>max receive size</code>	<code>(integer)</code>
<code>max receive number</code>	<code>(integer)</code>

Figure 3-2, Task Information Storage Format

The last major procedure is `write_file`, and its function is to use the `task` structure to generate two temporary output files for the DCL program and the software modeling stage. The first file is “`task_names.dat`,” and it contains a simple listing of all the task names in the suite. The second file is “`list_of_tasks.dat`,” and it is a listing of all the information contained in `task`, with one line devoted to each task instantiation and its associated data.

The other procedures used by `START.C` are `extract_name`, `get_rg`, and `strip`. These are minor functions needed for gathering data from the individual lines of the task specification file.

3.4 Flow of Execution

`START.C` is a simple series of procedure calls to extract information from the task specification file, organize it, and pass it on through temporary files. It begins with a call to `read_list`, and this is followed by a filename inquiry. The task specification file is opened, and then an iterative search process begins. `get_line` is used to grab successive lines from the file, and each call to `get_line` is followed by a call to `search`. Thus the file is examined one line at a time until `get_line` signals that the

end of the file has been reached. All information gathered is placed in the structure `task`, and `write_file` is invoked to produce the two output files described above. Figure 3-3 illustrates the flow of input and output files for the preliminary processing stage.

3.5 The DCL Program

FIND.COM is an elementary file search program written in Digital Command Language (DCL). Its purpose is to find the files which contain the source code for each application task in the task suite. There is no convention which demands that the filename in any way relates to the name of the task; also multiple tasks could be found in a single file. Of course, it is assumed that the user already knows where the source code can be found, but for purposes of automation, FIND.COM is utilized to avoid having the user enter the filenames for the various application tasks.

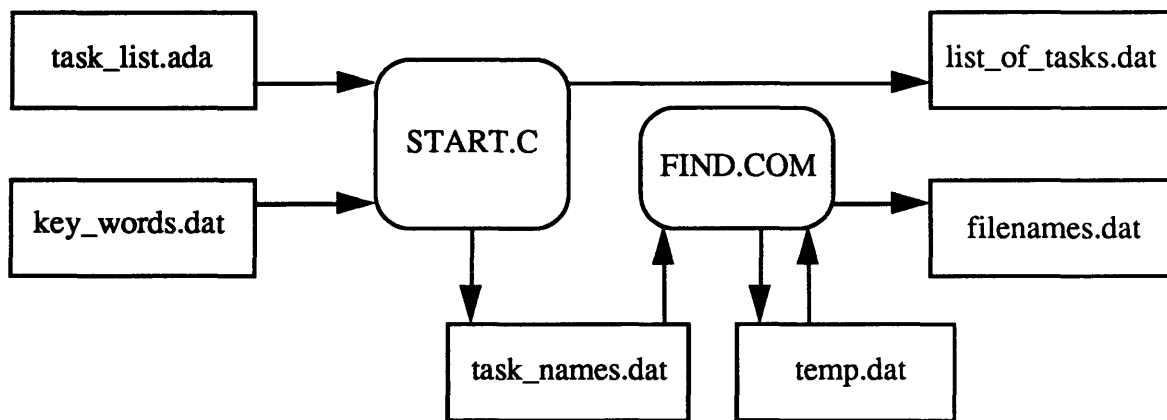


Figure 3-3, Preliminary Processing File Flow

The Digital Command Language has a powerful search command which allows a programmer to specify a word or group of words and then search all files in any number of directories to find occurrences of that word group. FIND.COM is programmed to search a specified set of directories where source code should be located. It first searches for the occurrence of the phrase “task body is;” this phrase signifies the presence of task source code. All filenames containing this phrase are stored in a file called “temp.dat.” This is followed by a second search in which the previously identified files are again searched for the individual task names listed in “task_names.dat.” When a file is found to hold a specified task, the task name and filename are recorded together in an output file called “filenames.dat.” The interaction of input and output files is shown in Figure 3-3.

In conclusion, the preliminary processing phase is relatively simple, but it really is essential to the analysis because the hardware and software modeling depend upon the configuration data. The details of pre-processing are not critical to the analysis; what is important is an understanding of the origin, motivation for, and contents of the two output files -- "filenames.dat" and "list_of_tasks.dat."

Chapter 4

Software Analysis

4.1 Assumptions and Limitations

The software analysis is the second stage in the progression of the AFTA timing analysis; its function is to examine the source code for the application task suite, develop a parameterized model for each task individually, and pass that model on to the hardware analysis stage to produce the desired timing results. The ultimate goal of the software analysis is to define a worst case bound on execution time for any single cycle of an application task. Recall that this information is necessary for the prevention of dynamic performance failures as described in Chapters 1 and 2. The difficulty of predicting software execution delay grows as the complexity of the code increases. High level languages such as Ada give the programmer a great deal of latitude in formulating a task's structure and flow of execution, and the problem of predicting execution time through a static analysis of source code is almost unmanageable. For this reason, some simplifying assumptions must be made, and the goals of the software analysis must be further defined.

The fundamental assumption for the software analysis is that task execution time can be accurately modeled according to a limited set of known functions, which are called in the course of code execution. At this point in the AFTA's development, the set of known functions primarily consists of benchmarked operating system calls [CLAS92], but in the future, any commonly used function can be benchmarked and added to the model. The source code analysis seeks to parameterize a task by determining how many times each of the known functions is called during a single execution cycle under worst case conditions. Since the delay for each function can be carefully designed and benchmarked to be a deterministic quantity, some simple algebraic manipulation is used to arrive at an estimated lower bound on worst case task execution time.

Notice that the timing analysis produces a lower bound on worst case delay, not an upper bound as one might expect. This is due to the fact that the task source code is modeled as an accumulation of function calls, where each call adds a known delay to the total worst case execution time. The set of functions included in the code model does not account for all of the processing performed by a task; this is clarified in the explanation of the code model found in Chapter 5. It is probable that a substantial amount of processing activity within the task could be overlooked by the model if such activity does not qualify as a known deterministic quantity. This is why the timing analysis can only produce a lower bound on worst case execution time; an upper bound would have to account for all

processing activity, and that is not possible for the code model developed in this thesis. The objective here is to construct a flexible model for source code that will produce increasingly accurate results as the model develops and becomes more sophisticated. Though it is not known exactly how a single task's execution times may be distributed, Figure 4-1 shows an exponential distribution as a reasonable possibility. Given this assumption, it is the goal of the software analysis to produce a worst case estimate that falls on the far right side of the curve. Since this analysis provides only a lower bound on worst case delay, the estimate will never be on the extreme right, but as the AFTA benchmarking efforts proceed and the code model grows and improves, the delay estimate should shift significantly toward the tail of the curve.

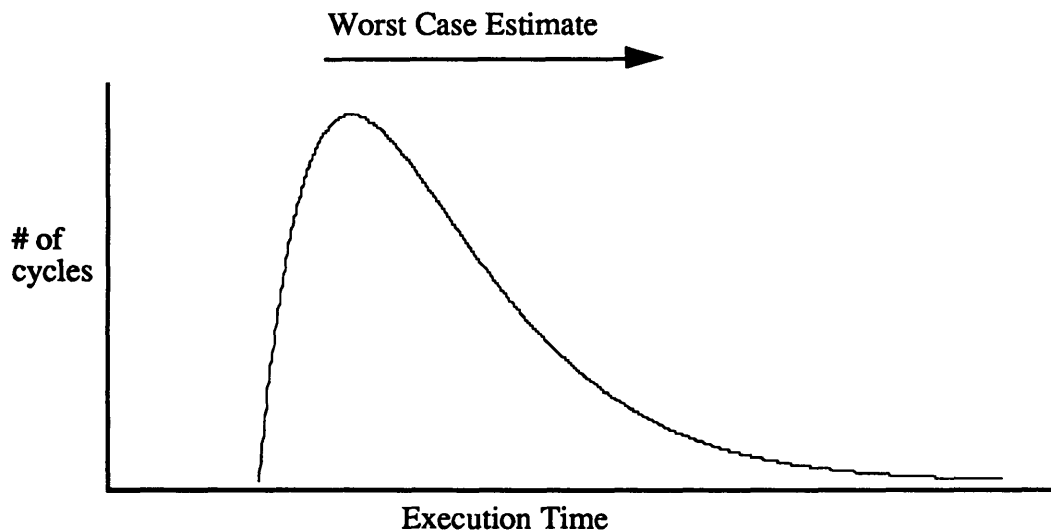


Figure 4-1, Possible Distribution of Execution Times

The software analysis works with Ada source code, and it is assumed that the code has been compiled and is free of errors. The analysis is heavily dependent upon Ada syntax rules in extracting correct information from the source code, and in fact, compliance with Ada syntax rules is the only guarantee afforded to the analysis tool when it encounters a segment of code. All programmers develop their own style and use unique text formatting and code structuring. For this reason, it is imperative that the software analysis is able to understand any legal code construct, rather than being tuned to accept a predefined code format. This allows full flexibility to the application task programmer, and at the same time, it makes the software analysis quite complex.

The element of style flexibility also imposes some limitations on the effectiveness of a static code analysis, for there are instances where it is useful to know the value of a variable, but because of the variable's definition and the flow of execution, a static

analysis is not able to define that value. In such cases, the analysis relies upon extraneous information from the programmer in the form of comments, and if these comments are not included, default values must be assumed. The individual programmer makes the decision whether or not to include the extra information, and if he chooses not to do so, the analysis is dependent upon default values which could be highly inaccurate.

Another limitation of the software analysis arises from the fact that there is currently a lack of actual application task code available for the AFTA. Since the AFTA is only a prototype at this stage, software development efforts are primarily focused on the operating system, with application task development being deferred. This serves to limit the amount of operational testing that can be done with timing analysis. Certainly, as a greater quantity and variety of task code is made available for testing, the timing analysis will be modified and improved. The fact that this analysis tool must be able to comprehend code that is not yet written reinforces the concept of allowing the applications programmer full latitude in the realm of style and assuming nothing about the source code except Ada syntax compliance. To date, the analysis tool has only been tested on the fault detection and isolation software written for the AFTA.

4.2 The Ada Software Structure

The Ada programming language is designed specifically for large, real-time, embedded computer systems, and it was created as a Department of Defense standard for software engineering. As such, Ada was chosen to be the language for software development on the AFTA [AFTA91]. An intimate knowledge of the Ada programming language is not necessary for one to understand the AFTA timing analysis, but a simplified understanding of the code framework is useful.

4.2.1 Subprograms and Packages

Ada code structures can be divided into three primary types of program units: subprograms, packages, and tasks. These program units generally have a two-part structure consisting of a specification and a body. The specification details the interface to the unit, and the body contains the unit implementation details which can be logically hidden behind the interface [BOO87]. The AFTA software analysis is interested in only the program body; the specification is generally ignored.

Ada subprograms are the basic units of execution, and the body of a subprogram holds the sequence of statements that define some algorithm. Subprograms are divided into two classes: functions and procedures. The software analysis deals primarily with

procedures, but the methods presented in this thesis are fully applicable to functions as well.

An Ada package is a unit of encapsulation which allows the programmer to group logically related entities such as subprograms and tasks. The software analysis relies upon examination of package bodies to find source code for application tasks and relevant procedures. Note that task source code is always enclosed within a package.

4.2.2 Tasks

Ada's real-time processing capabilities are based upon the use of program units known as tasks. A task is defined as an entity that can operate concurrently with other program units [BOO87], and the central concern of the software analysis is the examination of application task source code. A rate group task must have a well-defined cyclic execution behavior, and it should never reach a point of termination until system shutdown. These characteristics imply an infinite loop structure for a task, and this is shown in the following sample code fragment.

```
with scheduler;

package body appl_test is
task body appl1_t is
    my_cid : constant communication_id_type := appl1;
    num_deleted : natural := 0;
    frame_time : time := startup_time;
begin
    loop
        scheduler.wait_for_schedule;
    end loop;
end appl1_t;
end appl_test;
```

Figure 4-2, Sample Rate Group Task

Figure 4-2 illustrates the general code framework an application task. The statement `with scheduler` signals the inclusion of all subprograms in an external package called `scheduler`. This is followed by the definition of the package body for `appl_test`, which includes the body for a single application task known as `appl1_t`. The definition of `my_cid` is necessary for intertask communication, and `num_deleted`

refers to the number of messages deleted in the previous frame due to inadequate buffering. The variable `frame_time` contains the value of the time when the current rate group frame was started. These variables are unimportant to the software analysis; what is critical to note here is the structure of task `appl1`. Figure 4-2 shows `appl1` as a minimal rate group task consisting of an infinite loop surrounding a single call to the `wait_for_schedule` procedure that is defined within the `scheduler` package. The infinite loop ensures that `appl1` executes in an iterative manner for an indefinite length of time, and the call to `wait_for_schedule` allows `appl1` to suspend itself after each progression through the loop. The task is revived when scheduled by the rate group dispatcher during its next rate group frame. Thus, the `wait_for_schedule` call regulates the rate group behavior of a task in the following manner:

1. A task begins execution when scheduled by the rate group dispatcher.
2. When a task finishes a single execution cycle (in this case, one progression through the infinite loop), it calls `wait_for_schedule`.
3. The task then enters a state of self-suspension, and its processing resources are freed for use by other tasks.
4. When the next rate group frame begins, the rate group dispatcher schedules the task for another execution cycle.
5. Once scheduled, the task begins execution with the code immediately following the last `wait_for_schedule` call and proceeds until it encounters another `wait_for_schedule` call.

The progression of events described above is the basis of the software analysis.

4.3 Requirements of the Programmer

One of the goals of the software analysis is to be able to accept and comprehend source code written in almost any programming style, provided that the code complies with Ada syntax rules. In general, the programmer is not restricted to following any pre-defined format for the benefit of the timing analysis; however, there are a limited number of simple directives that, if followed, make the analysis much more effective.

4.3.1 Comment Information

When one performs a manual static analysis of source code, it is quite easy to scan the code both forwards and backwards to extract the information needed to trace the flow of a program. The human mind is capable of performing complex searches for variable values through many levels of definition, and one can almost mentally simulate source code as he reads it. Unfortunately, such a manual analysis is impractical and prone to

errors when dealing with exceptionally large bodies of code; thus, in the interest of speed and accuracy, automated analysis replaces manual analysis. The problem is that it is a difficult task to train a computer to read and understand source code the same way that a human would, and in order to simplify this task, some compromises must be made.

The first compromise is to force the computer to read code in only the forward direction. Certainly it is possible to have the computer search a piece of code in both directions to find information, but the logic to control such a search involves unjustifiable complexity. Therefore the AFTA software analysis reads and examines source code one line at a time in the order in which it is encountered. Any information that could be useful later in the analysis must be properly extracted and stored (or remembered) because there is no way to refer back to code that has already been processed. This approach may seem to be too limited, but the implementation of some simple strategies make it surprisingly effective.

One limitation of single-line processing is its inability to fully trace variable values. This is easily demonstrated with the following example:

```
task body example_t is
  counter : natural := 50;
  i : natural;
begin
  loop
    for i in 1..counter loop
    end loop;
    scheduler.wait_for_schedule;
    counter := 100;
  end loop;
end example_t;
```

Figure 4-3, Variable Tracing Example # 1

Now look at the code in Figure 4-3 from the computer's perspective -- read only one line at a time and do not refer to previously read lines. The computer sees the variable `counter` take on an initial value of 50, and then it finds a `for .. loop` that is iterated 50 times during the first execution cycle of the task named `example`. When the task is scheduled for its second cycle, `counter` assumes a value of 100 and the `for .. loop` is now executed 100 times. Unfortunately though, the computer has already associated a value of 50 iterations with the inner loop and cannot go back and find out how the new

value of `counter` affects previously processed code in subsequent execution cycles. A simple solution to this problem might be to require the computer to remember that the variable named `counter` affects the inner loop so that when `counter` changes, a new iteration value can be associated with the inner loop. This sounds simple until one considers that the value of `counter` could change within a deeply nested procedure, and the attempt to follow a variable into and out of a nest of procedures would require some very complex logic. Such a process borders on simulating code rather than performing a static analysis, and code simulation really has no advantage over actual code execution. Another solution would be to just dismiss single-line processing as an inadequate approach because of such a shortfall, but before doing so, consider the following example:

```
task body example_t is
  counter : natural;
  i : natural;
begin
  loop
    counter := temperature (format => kelvin);
    for i in 1..counter loop
    end loop;
    scheduler.wait_for_schedule;
  end loop;
end example_t;
```

Figure 4-4, Variable Tracing Example #2

The example in Figure 4-4 illustrates the possibility that a loop control variable could be determined by some quantity that is defined only at run time. In this particular case, the variable `counter` is a function of the temperature measured by some external sensor. No form of automated analysis can properly define a value for `counter`, and instances such as these require that the programmer provide some extra information to help make a static code analysis effective. Recall that the AFTA timing analysis is interested in worst case scenarios, and for this example it would be helpful to know that a value such as 500 is a reasonable limit on the value of `counter`. Only a programmer with extensive knowledge of the system could provide such input; an automated analysis tool cannot be expected to bridge such information gaps.

The first example illustrates a situation where programmer input is not required but serves to simplify the static analysis process, while the second example demonstrates that there are certain situations in which programmer input is absolutely necessary to make the static analysis effective. If the programmer uniformly provides information about loop iteration maximums, he guards against the analysis roadblock presented by the second example and at the same time provides a simplified solution to the variable tracing problem of Example #1. Situations of both types arise during a static code analysis, and a simple request that the programmer provide information about maximum loop iterations ultimately makes any static analysis more useful and actually establishes single-line processing as a viable approach.

The AFTA timing analysis benefits from four types of extraneous information that the programmer should be able to provide. They are as follows:

1. What is the maximum number of iterations in a for..loop?
2. What is the maximum number of iterations in a while..loop?
3. What is the maximum number of bytes in a message passed between tasks?
4. What, if any, is the maximum number of iterations for a basic loop?

This information must be provided in a conventional format that is convenient for the programmer to understand and easy for the analysis tool to read. The most appropriate method to communicate such data is through comment lines within the source code. These are the rules governing the use of comment information:

1. The information must be conveyed using a single comment line which begins with the standard "--" format.
2. This is followed by a "*" and an identifier denoting the type of information.
3. Next there is a statement of the form "max=" followed by a base 10 integer or the word "undefined" for infinite loops or unknown values.
4. The comment information must precede the corresponding loop or message, although it does not have to be placed immediately before the code statement.
5. To avoid confusion with constructs such as nested loops, information relating to an inner loop must fall within the adjacent outer loop.
6. Refer to Figure 4-5 for some examples.

The concept of asking for extra information from the programmer is not unreasonable or uncommon. In [PARK90], the authors describe a requirement for both maximum and minimum loop bounds for input to their program timing tool. Similarly, [PUSC89] discusses alterations to the original programming language for the specification of loop maximums in terms of an iteration count or a time delay. The

comment information method described here has the advantage of being simple to implement and flexible with regard to expansion. Its primary disadvantage is that there is no compiler enforcement of this convention. The programmer is free to omit the information, or if he chooses to include it, he might use an improper format.

```
-- * for loop: max = 100
-- * while loop: max = 200
-- * message: max = 300
-- * basic loop: max = 400
-- * basic loop: max = undefined
```

Figure 4-5, Examples of Comment Information

4.3.2 Naming Conventions

Early generation programming languages like FORTRAN and COBOL implement global naming conventions and burden the programmer with name space management. When naming an object, the programmer is forced to reference name listings to ensure that the name is used in the proper context and does not conflict with any other name in the system. Ada attempts to avoid this problem through the implementation of scope rules and the overloading concept. The objective is for code nesting and overloading to allow programmers to pick the most meaningful and convenient names for their objects without being concerned about the use of the same names in other parts of the system. The compiler sorts out the details in cases of name conflicts and thus gives the programmer a great deal of freedom. Unfortunately, the AFTA timing analysis tool is not as smart as the Ada compiler, and it is necessary to impose a few simple naming conventions.

The first convention is that all task names end with “_t” when used to define the body of the task. The analysis tool finds the task names in the task specification list and automatically appends the “_t” when searching for the source code of the task body. If the task name is not appended in this manner, its source code will not be found and will not be evaluated. For an example task body definition, refer back to Figure 4-3.

The second convention is that all procedures and functions defined within a single task must have unique names. The reason for this is that the software analysis builds source code models based upon procedure calls, and a specific model is associated with each procedure name. If a task makes use of two procedures with the same name, the analysis tool will be confused as to which procedure model to use. Refer to Chapter 5 for

a more detailed description of code model construction. Ada scope rules allow the same subprogram name to be used for multiple procedures or functions within a nested code structure, and the compiler is forced to figure out which body of code is being referenced for each occurrence of the name. Likewise, the overloading concept permits the programmer to use identical names for similar functions and procedures, provided that the compiler can distinguish them according to the parameters included with the subprogram name. The logic to sort such details is quite complex and is not included in the AFTA software analysis. This naming convention is the result of a simple engineering tradeoff where the need for simplicity in the analysis tool outweighs subprogram naming freedom for the programmer.

4.3.3 Undesirable Constructs

For a hard-real-time system, it is desirable for the software to exhibit a predictable timing behavior so that task scheduling constraints may be satisfied, and a software timing analysis is usually interested in examining worst case scenarios to ensure that execution timing deadlines are always met. When dealing with a worst case scenario, a static code analysis can be easily confused by certain programming constructs and thus provide no insight into their timing behavior. In practice, such constructs may demonstrate perfectly acceptable behavior, but they have a tendency to cripple the effectiveness of any a priori analysis.

One undesirable construct for real-time software involves the use of recursion. As mentioned previously, the AFTA timing analysis builds source code models based upon procedure calls. If Procedure A calls Procedure B, the model for Procedure B is inserted into the model for Procedure A. If Procedure A calls itself, the attempt to build a model for Procedure A is like trying to sketch the reflection of one mirror appearing in another mirror. Recursion may be the most efficient implementation for certain algorithms, but its timing behavior is usually unpredictable or at least very difficult to define. The AFTA software analysis notes the presence of recursive constructs, but it does not attempt to model them or analyze them.

As with traditional recursion, the use of mutual recursion is an undesirable construct for real-time software. Mutual recursion refers to the situation where Procedure A calls Procedure B and Procedure B calls Procedure A. The problem of trying to accurately model such a situation is similar to the one described above, and the AFTA software analysis notes the situation but does not attempt to analyze such a construct.

Another dangerous construct involves the use of basic loops and while loops. In order for the timing analysis to be effective, the programmer must specify the maximum

number of iterations for each instance of these types of loops, or a call to the `wait_for_schedule` procedure must be included inside the loop. Even if a `wait_for_schedule` call is placed inside the loop, there is a possibility that the timing analysis could signal the presence of a potential infinite loop. Obviously, an infinite loop is an unacceptable possibility for a real-time task. Figure 4-6 shows an example of such an undesirable situation.

```
loop
  if FLAP_STATUS = FULL_UP then
    compute(ALTITUDE, AIRSPEED);
  else
    scheduler.wait_for_schedule;
  end if;
end loop;
```

Figure 4-6, Potential Infinite Loop

Notice that the loop shown above has no defined maximum iteration count but does contain a call to `wait_for_schedule`. Thus, one might assume that any single execution cycle eventually escapes this loop by reaching the `wait_for_schedule` call; however, the software analysis looks at the worst case scenario only. In this case, it is possible that the `wait_for_schedule` call is never reached, and an infinite loop ties up processing resources and breaks timing deadlines. The AFTA timing analysis notes this situation but does not attempt to analyze it.

Chapter 5

Abstraction

The primary challenge of the software analysis is to identify and parameterize the worst case execution path for each application task. An execution path is defined to be any sequence of statements encountered between two successive calls to the `wait_for_schedule` procedure; thus an execution path determines the task's activity for a single rate group frame. In order to identify the worst case path, all possible paths must be explored, and a valid method of path comparison must be employed. For a static analysis, the best way to identify execution paths is to model the flow of execution for a given segment of source code, and the software analysis utilizes the concept of abstraction to construct such flow models for each task instantiation in the task suite. Abstraction is a method of hiding details in order to simplify analysis, and it is ideal for achieving the goals of the AFTA timing analysis. Recall that the fundamental assumption of the software analysis is that task execution time can be accurately modeled according to a limited set of known functions; abstraction is a way to eliminate details and highlight the role of these known functions. The goal of abstraction is to examine the task source code in full and develop a model that preserves only the information necessary for parameterizing the task in terms of known deterministic functions.

5.1 The Abstraction Methodology

The process of building source code models is rather complex; thus, it is best to begin with a high level discussion of the abstraction methodology. Task source code is input to the software analysis as a stream of characters from a file. These characters are assembled into single lines of code, and the code lines are processed individually and sequentially. In the context of this analysis, a single line is defined as that which falls between successive semicolons. Abstraction serves as a filter that transforms highly detailed source code into a simplified flow model for execution path analysis. As each line of code is processed, relevant information is extracted from it, and the code itself is then discarded. Information that is considered relevant falls into two categories: flow control items and known deterministic functions. The first category is a closed set of Ada constructs that are used by the programmer to define the flow of task execution. These include loop constructs, if-then-else constructs, case statements, and `wait_for_schedule` calls. The second category is an open set of functions composed of subprograms whose execution delay is deterministic and has been measured as part of AFTA system benchmarking efforts. Presently, this set of functions primarily consists of

operating system calls and in particular, message passing functions. Figure 5-1 shows a full list of the critical constructs that are extracted from the source code in the process of model construction.

```
loop
exit
for..loop
while..loop
end loop
if..then
else
end if
case..when
end case
scheduler.wait_for_schedule
rg_communication.queue_message()
rg_communication.retrieve_message()
rg_communication.send_message()
rg_communication.read_message()
rg_log.rg_log_entry()
debug_trace.debug_log()
rg_dispatcher.io_utils()
```

Figure 5-1, List of Critical Constructs

Each program statement is searched for items belonging to this list. When a critical construct is found, it is appended to the end of the model along with any data associated with the construct, and the model thus becomes a reflection of the task source code with all unnecessary details removed.

5.2 The Motivation for Abstraction

Familiarity with the abstraction methodology makes it is easy to understand the motivation for source code modeling and the use of abstraction. The justification for this approach to software analysis is best summarized as follows:

1. The use of code models expands the power of single-line processing. Examining program statements one at a time and in sequential order is a very restrictive way of analyzing source code. With the abstraction method, single line processing is only an initial stage that serves to build the task model; it is

not responsible for code analysis. Thus, any static analysis limitations of single-line processing disappear once the model is built.

2. The source code model is stored as a collection of integers, and this makes it easy to analyze and manipulate in an automated fashion. The timing analysis deals only with the task model; it completely ignores the original source code once the model is built. This allows the analysis to be greatly simplified because it is not wrapped up in code interpretation or string manipulation, and simplicity is vital to the improvement and maintenance of the analysis tool.
3. The task model is constructed in a format that is ideal for identifying and comparing possible execution paths. In contrast, examining the source code itself for execution paths is an extremely complex task. The abstraction process is designed specifically to transform the source code into a format that allows the most efficient execution path identification, parameterization, and analysis.
4. Abstraction serves to expose the message passing characteristics of an application task because all the message passing procedures are included in the list of critical constructs. When all tasks are considered simultaneously in the hardware analysis stage, the analysis tool develops a picture of worst case global message passing activity. This is important in determining the timing behavior of the rate group dispatcher, which is executed as part of the system overhead in every minor frame.

5.3 Code Model Elements

The source code model for an application task is stored as a collection of integers. Integer manipulation is easily accomplished within a high level language like C; thus the format of the model lends itself to a simple and efficient implementation of the model analysis procedures discussed in Chapters 7 and 8. The model is a one-dimensional array of entries, and each entry is a set of five integer elements. These elements are outlined below:

1. **TYPE:** An integer value representing the type of critical construct that is found in the source code and recorded as a model entry. Every critical construct listed in Figure 5-1 has a specific integer associated with it and is recognized by the computer according to its numerical value. The constant definitions for the analysis tool are found in "header.h," and for purposes of code readability and maintainability, these constants are referred to by the names listed in "header.h" rather than their respective numerical values.

2. **VALUE:** Some of the critical constructs have a value associated with them, and this value is required for analyzing worst case scenarios. For instance, for all loop constructs, the VALUE element represents the maximum number of iterations for that loop. Loops whose behavior is undefined are assigned one of two constant values whose names best describe the nature of the loop. The names of these constants are UNDEFINED and INFINITE; their meanings are self-explanatory. Message passing constructs also have an associated value which represents the maximum size of a message in 64-byte packets. If no maximum value is specified by the programmer, the message passing limitations listed in the task specification file (refer to Chapter 3) are used as default values.
3. **DEPTH:** After the model is created, each entry is assigned a DEPTH value to represent its level within the nested structure of the source code. The DEPTH element helps the model emulate the modular construction of the original code, and the information conveyed by the depth element is vital to the task of identifying possible execution paths.
4. **POINTER:** Not to be confused with a pointer in C, the POINTER element is a value assigned to a model entry after the model is fully constructed. It is the index value of another closely associated model entry, and its assigned value is essential for proper identification of possible execution paths. For example, the `end loop` construct uses the POINTER value to identify the index for the model entry that represents the beginning of the loop construct. The specific rules governing the assignment of the POINTER value are explained in Chapter 7. Not all model entries require the use of this element, and in such cases, the POINTER element is assigned a constant value named UNDEFINED.
5. **FLOW:** This element is used only during the generation of execution paths through the code model. It is required for bookkeeping purposes, and its value assignment conventions are discussed in Chapter 7.

The figures on the following page illustrate a sample model construction. Figure 5-2 is a segment of test code designed to highlight the occurrence of critical constructs; it is not intended to represent any particular algorithm. Notice in the parameter specifications for the message passing procedures that only the fourth parameter is specified. This parameter indicates the size of the message; the other parameters are unimportant to the timing analysis and are not included. It is important to understand that the source code lists the message size according to bytes while the model stores the

```

loop
  for i in 1..20 loop
    rg_communication.queue_message(--,--,--,100,--,--);
    if VAR_1 > VAR_2
      rg_communication.queue_message(--,--,--,150,--,--);
    elsif VAR_2 > VAR_3
      rg_communication.retrieve_message(--,--,--,175,--,--);
    else VAR_2 := 0;
    end if;
    scheduler.wait_for_schedule;
  end loop;
  while (VAR_1 < 100) loop
    VAR_1 := VAR_3 + 5;
    scheduler.wait_for_schedule;
  end loop;
  rg_communication.queue_message(--,--,--,200,--,--);
end loop;

```

Figure 5-2, Sample Source Code Segment

INDEX	TYPE	VALUE	DEPTH	POINTER	FLOW
0	LOOP	(1) INFINITE	0	15	N/A
1	FOR_LOOP	(55) 20	1	10	N/A
2	QUEUE	(7) 2	2	UNDEFINED	N/A
3	IF	(2) UNDEFINED	2	5	N/A
4	QUEUE	(7) 3	3	UNDEFINED	N/A
5	ELSIF	(4) UNDEFINED	2	7	N/A
6	RETRIEVE	(8) 3	3	UNDEFINED	N/A
7	ELSE	(3) UNDEFINED	2	8	N/A
8	END_IF	(52) UNDEFINED	2	UNDEFINED	N/A
9	WFS	(0) UNDEFINED	2	UNDEFINED	N/A
10	END_LOOP	(51) UNDEFINED	1	1	N/A
11	WHILE_LOOP	(56) UNDEFINED	1	13	N/A
12	WFS	(0) UNDEFINED	2	UNDEFINED	N/A
13	END_LOOP	(51) UNDEFINED	1	11	N/A
14	QUEUE	(7) 4	1	UNDEFINED	N/A
15	END_LOOP	(51) UNDEFINED	0	0	N/A

Figure 5-3, Sample Source Code Model

message size according to the number of packets since all analysis calculations deal with packets rather than bytes. Figure 5-3 shows the code model that is constructed from the preceding code segment; notice that the flow element is deliberately left unspecified since it is only used during execution path generation.

The model follows the source code very closely since the code is primarily comprised of critical constructs. Each occurrence of a critical construct is represented by a single model entry. For purposes of readability, the TYPE element is listed according to the name of the appropriate constant; the actual constant value is shown in parentheses. The top entry indicates an infinite loop, which is followed by a for . . loop with a maximum of 20 iterations. The queue_message procedure call is included as the third model entry, and the message size (100 bytes/2 packets) is shown in the VALUE element. The if construct in entries 3 through 8 demonstrates one use of the POINTER element. When a conditional statement is encountered during code execution, alternate paths may be followed. The POINTER value indicates where execution continues if a given condition is not satisfied. For example, if the condition associated with entry #3 is not met, execution continues with entry #5, and if the condition there is not met, execution continues with entry #7. The POINTER value holds the key to following the proper path. Notice that the model includes no information about what conditions are imposed by the if statement in entry #3 or the else if statement in entry #5. The timing analysis is not intended to simulate the source code; rather, its intermediate goal is to find all possible execution paths. For this reason, the actual test condition is ignored; it is only important for the model to indicate that alternate execution paths exist due to the presence of a conditional statement. Following the if-then-else construct, entry #9 indicates a call to wait_for_schedule (WFS); a WFS call marks both the beginning and the end of any execution path. Entries #10, #13, and #15 illustrate the use of the POINTER element to link the end of a loop with the beginning of that loop. A final point of interest for this model is the use of the DEPTH element to indicate the nesting level for each entry. In Figure 5-2, the use of indentation shows the logical nesting of code statements. Likewise, the value of the DEPTH parameter in Figure 5-3 carries the same information. For any complete subprogram or task, the DEPTH value begins and ends at zero, and each level of nesting has a successively greater value.

5.4 Bottom-Up Construction

The power of high level languages springs from the use of modularity, and it is essential that a static analysis tool is able to recognize and properly interpret the use of procedures and functions within an application task or within a supporting subprogram.

When one manually performs a static analysis and encounters a procedure call, it should be easy to find the code for that procedure and examine it within the context in which it is called upon; however, such a searching task is not so trivial for an automated analysis tool. For an automated analysis, when a procedure call is found, the analysis is suspended while the computer searches for the source code belonging to that procedure. The source code could be in the file that is already open or it could be in a separate file, in which case two or more files are simultaneously left open for examination. The problem grows in complexity when one considers the possibility that the first procedure could call another procedure whose source code resides in yet another file. Of course, this type of procedure nesting can go on for many levels and would result in multiple open files and multiple suspensions of the analysis process. Note that each suspension of analysis results in a complicated effort to save present state information in a useful format. Obviously this is an unwieldy method of dealing with modular source code.

The use of a source code model avoids the problems inherent in the cyclical analyze and search method described above. The tasks of processing source code and analyzing source code are separated through the use of the code model. All the code processing is directed toward building the model for an application task, and then the model is analyzed independently with no further references to the source code. In order for the model to be accurate and effective, whenever a procedure call is encountered, the modeling tool must already know the name and nature of that procedure so that it does not have to search for its source code. Thus, the source code model must be constructed in a bottom-up fashion -- precisely the opposite of the top-down manner in which software is created. In other words, the modeling tool begins by examining the most elemental procedures and progresses upward to higher level procedures, which often call upon the more elemental procedures. Figure 5-4 illustrates this concept.

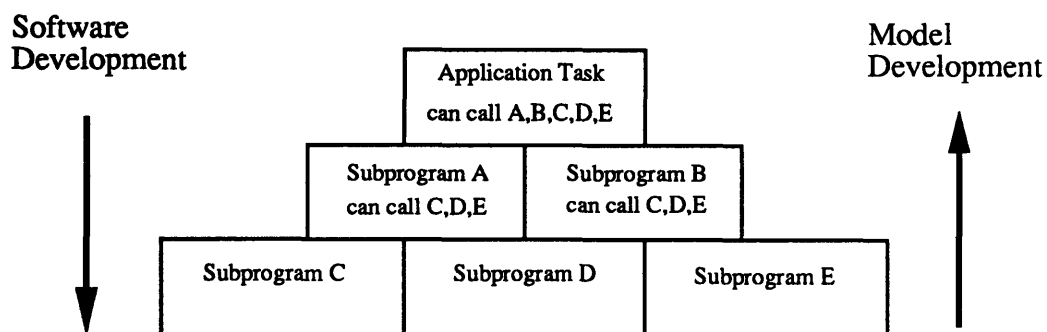


Figure 5-4, An Ideal Software Hierarchy

This type of bottom-up model development depends upon an understanding of the program unit linking mechanisms employed by Ada. An Ada compilation unit is defined as the specification or the body of a program unit, which can be compiled as independent text; the body of an application task is one example of a compilation unit. It may be preceded by a context clause that identifies other compilation units upon which it depends, and the context clause uses `with` statements to name the supporting program units [BOO87]. In general, the package containing a task body begins with a context clause that indicates which packages contain subprograms utilized by the task body, and this is the key to identifying the particular hierarchical structure of a task. A package referenced by the task's context clause may have its own context clause for the subprograms upon which it depends. Thus, an examination of all relevant context clauses reveals the task's dependency relationships and enables the model building process to begin at the bottom of the pyramid as shown in the preceding figure. Chapter 6 explores the procedure by which the software hierarchy of an application task is exposed and utilized during the modeling process.

5.5 Expandability

A key feature of the abstraction approach is the ability to later expand upon the code models that are generated and thus improve the results of the timing analysis. Current benchmarking efforts for the AFTA [CLAS93] are aimed at quantifying operating system overhead, and it is for this reason that the critical constructs list (Figure 5-1) is primarily composed of operating system calls. As the project progresses, system benchmarking will determine the execution delay for many more functions that could be used by the operating system and/or the application tasks. Once a function is measured, it will be added to the list of critical constructs, and its delay will be accounted for in the hardware modeling stage of the analysis. The timing analysis software is written in such a manner as to facilitate the addition of critical constructs to the system model, for there are a limited number of procedures affected by such an addition. Appendix G outlines the specific steps that should be taken to update the timing analysis software for additional critical constructs.

Chapter 6

Source Code Processing

The software analysis stage is divided into two phases: source code processing and task model analysis. The goal of the source code processing is to build a complete and accurate model of the application task source code. Once the model is developed, the source code is no longer needed, and the task model alone is presented for further analysis.

The primary objectives of source code processing are defined as follows:

1. To understand the program unit hierarchy for each application task and to know where to find every relevant unit of source code.
2. To properly interpret program statements and understand modular code construction regardless of any specific programming style in use.
3. To extract relevant information in an orderly manner, store it in an efficient format, and apply it to task model development.

6.1 The Big Picture

An explanation of the execution flow for source code processing is rather tedious, and it is helpful to first understand the basic structure of the timing analysis software. On the following page, Figure 6-1 illustrates the hierarchy of procedures used in both the software and hardware analysis. Notice in this diagram that subordinate procedures are physically linked to their parent procedure(s); these links should help demonstrate the context in which each procedure is called. The source code for all procedures shown is listed in `FINISH.C` and is included in Appendix D.

The source code processing progresses on a task by task basis, and the procedure called `task_parse` is primarily responsible for the model development of individual tasks. `task_parse` is called from `process_list`, and it is passed the name of a single task and the name of the file in which the task body is defined. Notice that `task_parse` is a pivotal node in the analysis hierarchy and essentially orchestrates all the activities associated with the software analysis. When `task_parse` is complete, it passes control back to `process_list` and also returns a parameterized representation of the original task.

6.2 Establishing the Hierarchy

The code processing begins with the development of the task's program unit hierarchy, and this is achieved by the procedure called `find_packages`. As Chapter 5

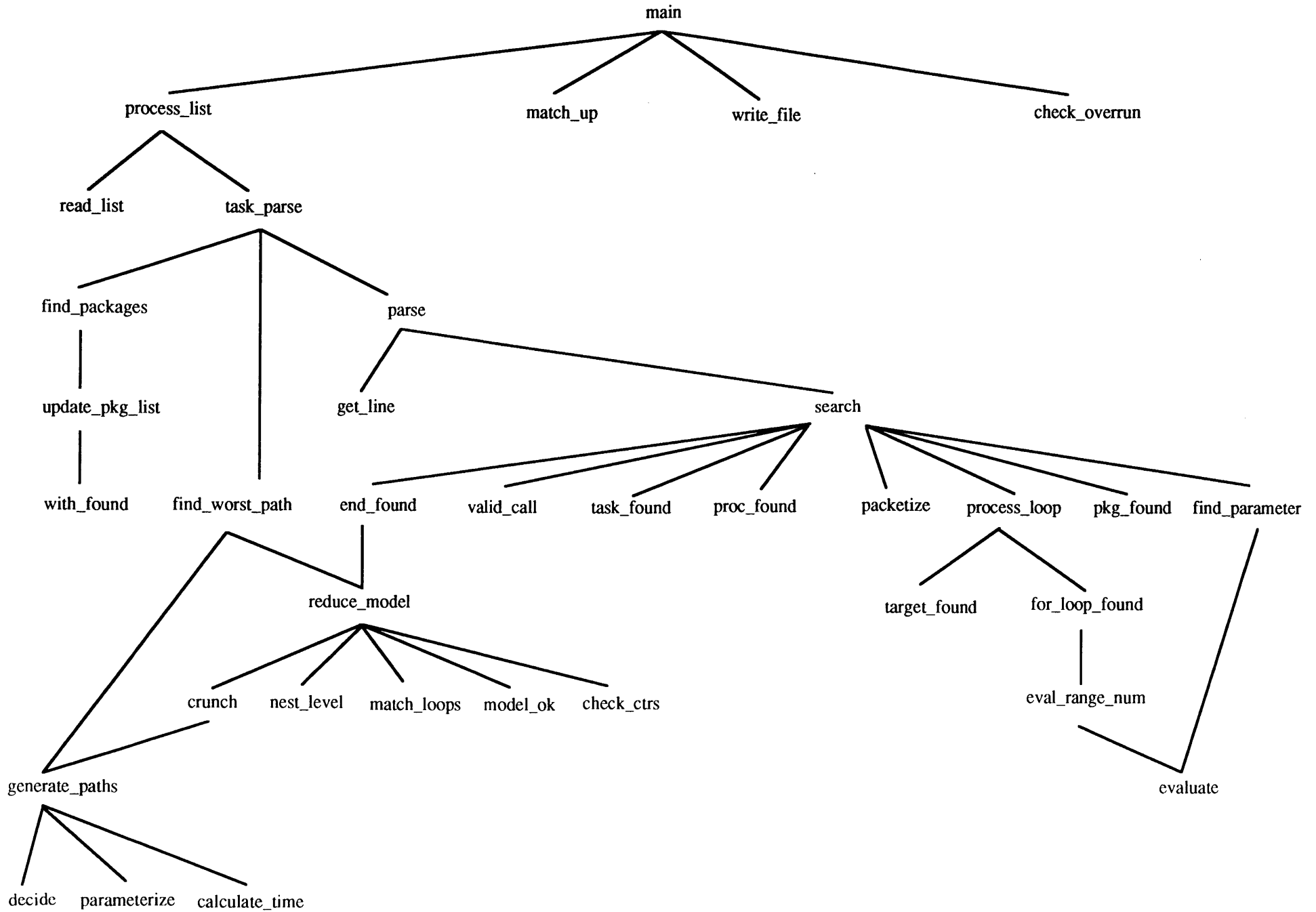


Figure 6-1, Timing Analysis Hierarchy

explains, application task software is typically organized according to package units whose names are specified in context clauses. `find_packages` is given the filename of the package containing a given task (as specified during preliminary processing), and it compiles a list of all packages containing subprograms used by that task and its supporting subprograms. This list is an array of filenames stored in a structure called `pkg_list`. The procedure `update_pkg_list` does most of the work for `find_packages`. It is given a filename and opens that file to search for the `with` statements of the file's context clause. For each `with` statement, it uses the procedure called `with_found` to extract the name of the associated program unit and then adds that name to the end of `pkg_list`, if it is not already present. The first time `find_packages` calls `update_pkg_list`, it sends in the name of the file containing the application task, and this begins the construction of `pkg_list`. On subsequent calls, `find_packages` extracts package names from `pkg_list` and converts them to filenames to be passed to `update_pkg_list` for processing. This process continues until the context clause for every entry in `pkg_list` is examined. In this way, all subprogram dependencies are explored in such a manner that the program units at the top of `pkg_list` are supported by the units found further down the list. No program unit should depend upon a unit that appears above it in the final package list.

6.3 Code Modeling Tools

Once `pkg_list` is fully constructed, the actual code modeling begins. `task_parse` starts by examining the units at the bottom of the list so that models are first developed for the most elemental procedures and functions. For each entry in `pkg_list`, `task_parse` utilizes the procedure called `parse` to perform the code examination and model construction. After each subprogram is modeled, any pertinent information is stored in a data structure called `procedures`. As the modeling process progresses, calls to previously modeled subprograms are encountered, and when this occurs, the model for the subordinate subprogram is inserted directly into the growing model of the parent subprogram. Once each package unit in `pkg_list` is processed by `parse`, the package containing the task body is sent to `parse` so that a complete model of the task is developed for later analysis.

6.3.1 `parse`

The `parse` procedure is a generic unit designed to process any type of source code, whether it belongs to a package, a subprogram, or an application task. It is used by `task_parse` as a black box processing tool that directs the low level parsing activity of

bottom-up task model development. The algorithm employed by `parse` is a simple iterative process as is shown on the following page in the flow diagram of Figure 6-2. Basically this procedure opens a designated file, grabs individual program statements with a procedure called `get_line`, and examines them with a procedure called `search`. The parsing process ends when any one of three conditions is satisfied. These conditions are as follows:

1. A fatal processing error occurs.
2. The task model is complete.
3. The end of the file is reached.

The inputs to `parse` describe the context in which the code processing is taking place, and these include items such as the name of the current task, the name of the file to be opened, the name of the package being parsed, and the message passing default values for the current task. The outputs from `parse` include an updated version of the `procedures` data structure, and if the task body itself is submitted to `parse`, the procedure outputs a complete model of that task. The operation of `parse` depends upon three key procedures: `read_list`, `get_line`, and `search`.

6.3.2 `read_list`

One of the initial functions of `process_list` is to call `read_list` in preparation for the parsing activity to follow. `read_list` is designed to read a pre-defined file called `key_words.dat` and extract from that file a list of key terms that act as a guide to interpreting source code. The list is stored as an array of string variables in a structure called `search_list`. Also stored in `search_list` is an integer called `length` to specify the number of items found in `key_words.dat`. This version of `read_list` is identical to the one described in Chapter 3, Preliminary Processing. The `search_list` structure really could be hard-wired into the timing analysis source code, but the use of `read_list` encourages code modularity and maintainability since the list of key terms can be altered without changing the source code for the analysis tool and forcing a recompilation.

6.3.3 `get_line`

The purpose of `get_line` is to pull individual characters from a specified input file and construct a single program statement that is stored in a structure called `this_line`. `this_line` is formatted as an array of string variables with each string representing a single word within the program statement. It also has an integer called `length` to describe the number of words in the program statement and an integer called

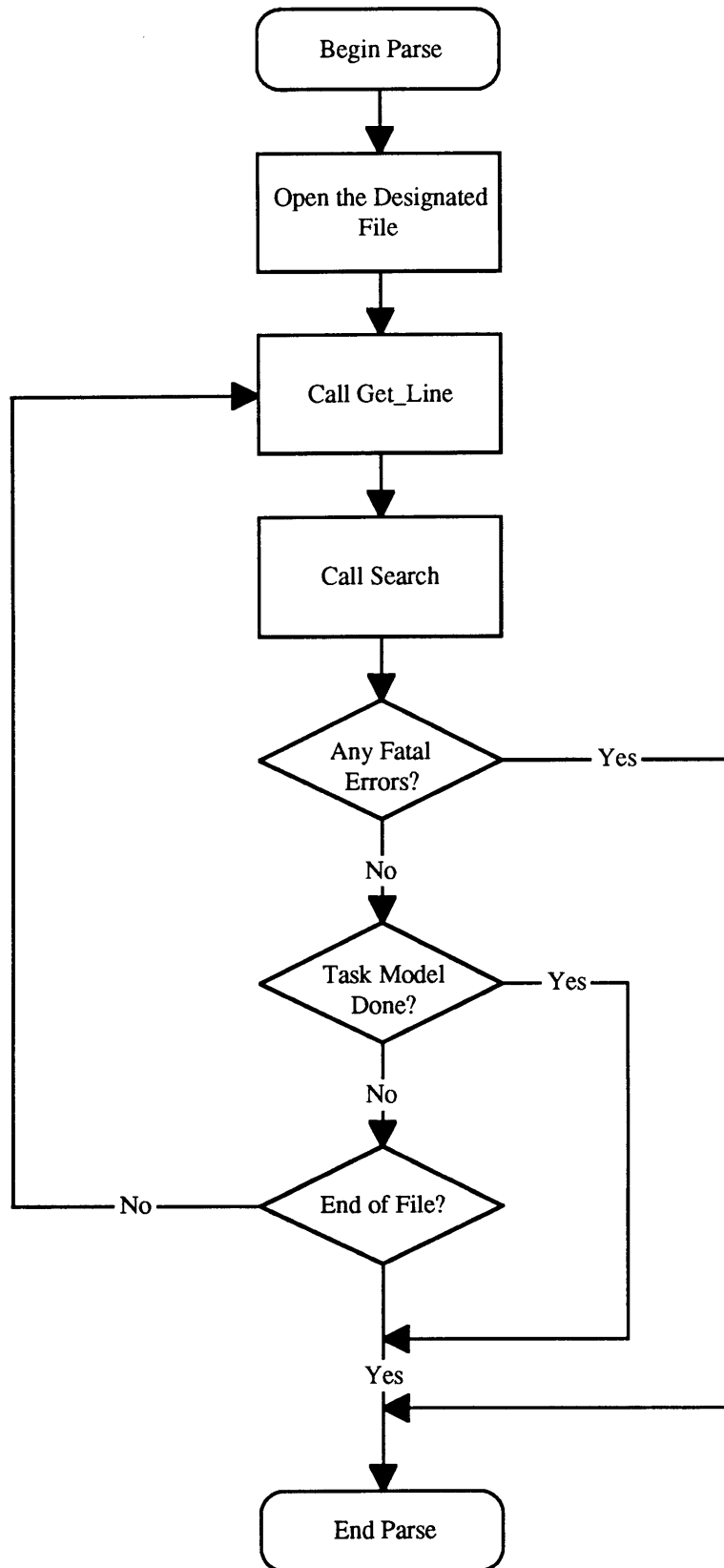


Figure 6-2, The parse Algorithm

`marker`, which can act as a pointer to a specific word of the program statement when `this_line` is passed between procedures. One function of `get_line` is to eliminate all white space and unnecessary comments from the source code and reduce a code segment to an unformatted series of program statements. The removal of indentation formatting from the code forces the code processing to ignore that aspect of programmer style. `get_line` also reduces all upper case characters to lower case characters as they are read from the input file; this makes the source code processing case independent and removes yet another element of programmer style.

6.3.4 `search`

A procedure named `search` is responsible for finding and extracting important information from the individual program statements, and it simultaneously directs the development of the current code model. See Figure 6-3 on the following page for a flow diagram of `search`. The current code model is stored in a structure called `skeleton` (since it is a bare bones representation of the source code). Both `search_list` and `this_line` are inputs to `search`, and for each program statement, this procedure compares every word to every entry in `search_list`. When an exact match is found, the current code model is usually updated, and part of the update may require that additional information is extracted from the program statement or from other data sources. For instance, if a message passing call is found, `search` activates a procedure called `find_parameter` in order to determine the maximum size of the message involved. Likewise, for any looping construct that is found, `search` uses a procedure called `process_loop` to determine the type of loop and the maximum number of iterations associated with that loop. `search` is essentially a grand `switch` statement with a single `case` entry corresponding to each item in `search_list`. The use of the `switch` statement allows the overall comparison process to remain generic while preserving the uniqueness of response for matches with various items in `search_list`.

Keep in mind that the code processing algorithm is intended to be as generic as possible so that the `parse` procedure can be used to deal with any segment of code, whether it belongs to a task, a package, a procedure, or a function. This approach does involve a simple tradeoff, however. The advantage is that only one set of searching procedures is written, tested, and maintained. The disadvantage is that the code examination process must be flexible enough to handle all situations and intelligent enough to recognize the source code from different types of program units. In other words, there is no master procedure that recognizes what type of code is being processed and selects the appropriate searching procedure; rather, all the processing decisions are

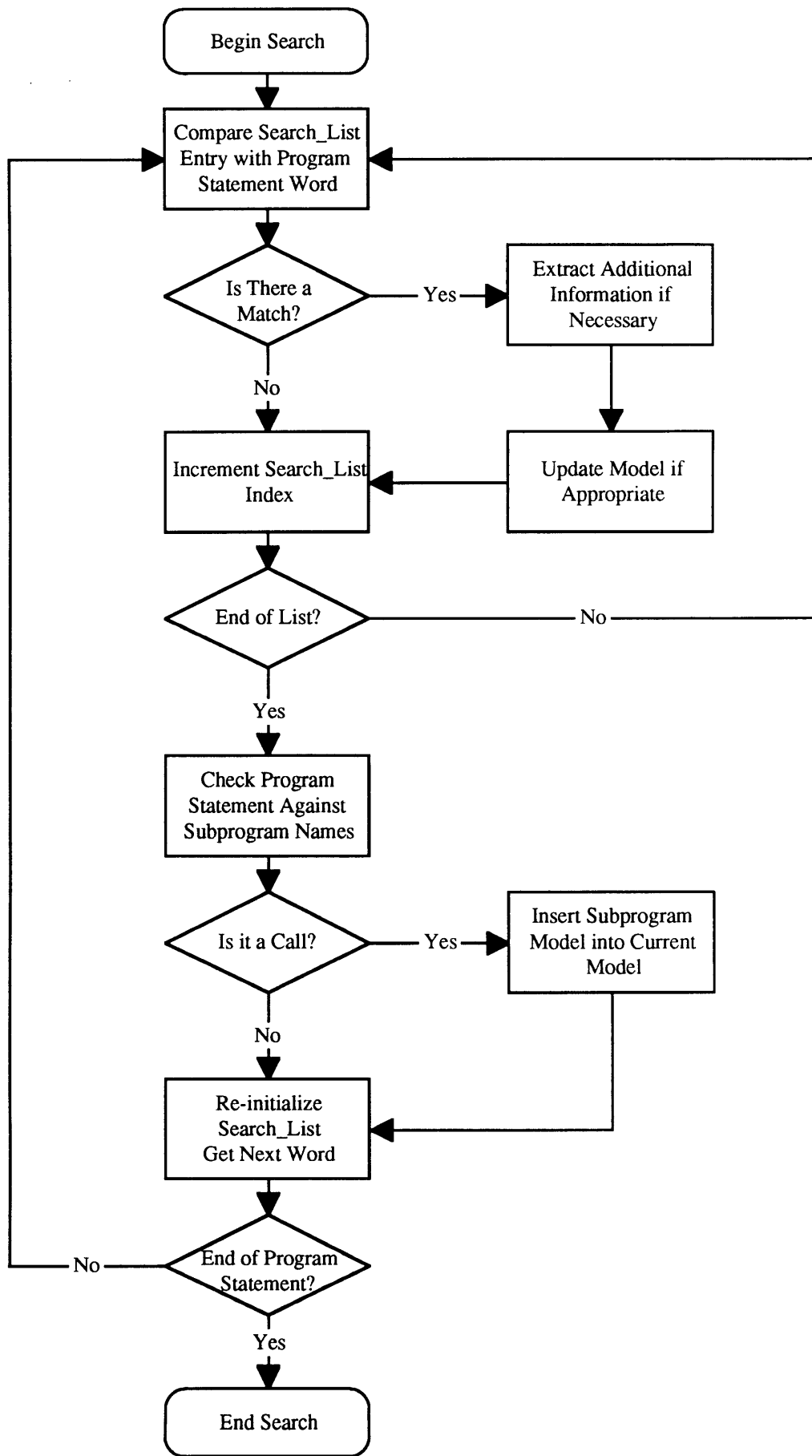


Figure 6-3, The search Algorithm

made at a low level as the search progresses. One consequence of this approach is that some overhead state information must be carried in and out of the search procedure through the passing of pointers in the parameter list. This information is needed by `search` in order to understand the context in which the current program statement is written. Every time `search` is called, it receives a single program statement along with pointers to information about the state of the current search process. The `search` procedure continuously gathers and maintains this state information to enable it to understand program statements that it will have to examine in the future.

Two variables that hold state information are strings called `task_name` and `pkg_name`. Recall that `task_parse` processes code in two stages: the first stage examines and models all supporting subprograms and the second stage examines and models the task code itself. The task name is needed only in the second stage when the task code itself is being processed; otherwise, it is defined as “none.” This differentiation alone tells `parse` whether it is dealing with task code or supporting subprogram code. The reason the name is necessary for task processing is that the source code for several tasks may be included within the same file, though tasks are analyzed only on an individual basis. Preliminary processing identifies the file containing a particular task, and when it is time to process that task, the appropriate file is opened. If there are multiple tasks within that file, the source code processing is applied only to the specified task; other source code is ignored. The only way that the code processing algorithm knows which code to examine and which code to ignore is if it can find the `begin` and `end` statements for the specified task. The `begin` statement always contains the task name, and the `end` statement typically uses the task name as well, although it is not required. Refer back to Figure 4-2 for a specific example. Thus the `task_name` variable is used by the code processing algorithm as a guide to determine where the task body definition is located within the file that is currently being examined. If `task_name` is defined as “none,” `parse` knows to process all source code and not to search for only the executable code of the task body.

The variable called `pkg_name` is needed for proper identification of subprograms. Once a procedure or function is examined and modeled, its model is stored in the structure called `procedures`, and a name is associated with it for identification purposes. When a subprogram is called from outside of the package within which it is defined, the appropriate package name precedes the name of the subprogram. For example, the procedure `wait_for_schedule` is defined within the package called `scheduler`, and a call to this procedure from the task body uses the name `scheduler.wait_for_schedule`. Thus, for every subprogram model, the name is

stored in the “package_name.subprogram_name” format so that when the subprogram call is encountered by `search`, the contents of the program statement exactly match the name of the model stored in `procedures`.

The name of the file currently being processed is also passed to the `search` procedure, and this is necessary for error tracking. When a critical error occurs in the model building process, the package name, procedure name, and filename are all included in the message recorded in the error file. Such error tracking is particularly helpful in developing and testing the timing analysis program, but it should also be helpful to the user in finding and correcting any undesirable constructs that are detected in the course of the timing analysis.

Also included in the parameter list of `search` is a pointer to the structure called `end_list`. This is a one dimensional array of strings that is useful in tracking the nesting structure of the source code. Ada syntax requires that many constructs conclude with an `end` statement, and a list of these constructs is included below:

```
package body [name] is.....end [package name];
task body [name] is.....end [task name];
procedure [name] is.....end [procedure
name];
function [name] is.....end [function name];
loop.....end loop;
if.....end if;
case.....end case;
record.....end record;
begin.....end [block name];
select.....end select;
```

Figure 6-4, Ada’s Framed Constructs

The ability to properly interpret a given program statement often depends upon knowledge of previous program statements, and this is particularly true when dealing with `end` statements. The maintenance of the `end_list` structure gives the code processing tools critical information about the context in which a specific program statement is written, and it also helps keep track of the nested structure of the source code. Whenever the beginning of a framed construct is detected, a corresponding string entry is added to the tail of `end_list`. Then, when an `end` statement is encountered, it is compared with the last entry in `end_list`, and the two strings should match. If they

do not match, a processing error has occurred and is properly noted in the error file. For instance, if the beginning of a loop is found, the string “end loop” is appended to `end_list`, and the code processing tools then know to expect that the next end statement encountered will be an “end loop.” This seems to be quite trivial for statements like “end if, end loop, end case,” and “end select” because the program statement explicitly defines the type of construct to which the end statement belongs. However this tracking process is absolutely necessary for the other constructs whose end statements remain rather vague. The end statements for packages, tasks, procedures, functions, and block statements do not have to include the name of the construct, though many programmers prefer to include the name for code readability and debugging purposes. When the code processing tools encounter a non-specific end statement (with no name included), there is no way to determine what program unit or construct is affected, but by checking the last entry in `end_list`, it is easy to interpret the meaning and significance of the end statement. This is illustrated in the following example:

```
package body ambiguous is
task body example_t is
  procedure compute is
    begin
      ...
    end;
  begin
    ...
  end;
end;
```

Figure 6-5, Example of end Statement Ambiguity

After processing the first three program statements shown in Figure 6-5, the `end_list` data structure has the following three entries: “end ambiguous, end example_t, end compute.” When the first end statement is found, the analysis program is uncertain whether that statement belongs to the package, the task, or the procedure, but after referring to the last entry in `end_list`, it finds that the end statement belongs to a construct called `compute`. The strings called `pkg_name` and `task_name` hold the names of the current package and task, and the structure called `procedures` holds the name of the current procedure. The states of these variables are

checked to find out that `compute` is the name of the current procedure. This is valuable information since it indicates that code processing is complete for the procedure called `compute`, and its model is ready to be sent to the model reduction tools (Chapter 7) for final processing. Once the first `end` statement is properly deciphered, last entry in `end_list` is eliminated, and two entries remain. The interpretation process is similar for the next two `end` statements, and when processing is complete for the code segment of Figure 6-5, `end_list` is left empty since the code processing properly works its way into and back out of the nested source code. If `end_list` is not empty when the end of a file is found, a fatal error has occurred and is so noted in the error file.

Yet another useful state variable needed within the `search` procedure is a structure called `flags`. This is a set of five boolean variables used to maintain and transfer status information during code processing. Each of the flags performs a necessary function although the types of functions vary widely. The set as a whole is actually an ad hoc compilation of status variables that is assembled to make it easy to pass different types of information between procedures under a single pointer name. All flags are initialized to a value of “no” at the beginning of the source code processing for each task, and the current values of the flags help the analysis program to make critical decisions about code processing and model development. What follows is a detailed description of the function of each of the five flags.

1. `model_active`: The executable code for a program unit generally follows some declarative statements such as variable definitions. There is no need to include these declarative statements in the model building process, and the model is considered to be inactive until the executable code is encountered. The executable code is bound by `begin` and `end` statements, and when the `begin` statement is found, `model_active` is given a “yes” value. Likewise, when the `end` statement is found, `model_active` is given a “no” value.
2. `task_found`: A file may contain more than one task body definition. It is therefore useful to know when the code for a specific task body is found so that code from other task bodies does not improperly contribute to the current task model. Once the statement “`task body [name] is . . .`” is found, the `task_found` flag registers a “yes” value. When the `end` statement for the task is encountered, `task_found` returns to its initial “no” value. Note that the `model_active` flag is dependent upon the `task_found` flag during parsing of the file containing the task body definition. The `model_active` flag cannot register a “yes” value until the `task_found`

flag indicates a “yes” value; thus the model building process begins only after the executable code for the appropriate task is located.

3. **pkg_found**: The software hierarchy for an application task is developed as a series of package names that are found within context clauses. The source code processing examines each package in succession while progressing upwards from the bottom of the package list. When `search` encounters the beginning of the specified package body, it sets `pkg_found` to a value of “yes,” and it likewise deactivates the flag when it encounters the end of the package. Neither the `task_found` flag nor `model_active` flag can be activated unless the `pkg_found` flag is already activated.
4. **finished**: A single file may contain several task body definitions, but tasks are processed on an individual basis. Therefore, it is reasonable for the software analysis to discontinue file parsing once the appropriate task body is examined and modeled. When the task body’s end statement is encountered, the `finished` flag is set to “yes,” and this leads to an exit from the parsing loop.
5. **fatal_error**: A fatal error occurs when the `end_list` construction encounters a mismatch. In other words, the source code processing might expect to find the end statement for a procedure but instead finds a statement like “end if.” Such an error indicates an interpretation mistake or oversight by the source code processing, and the current code model is labeled as invalid. This type of error undermines the effectiveness of the modeling and analysis for a given task because it indicates that part of the task model is incorrect, and thus the entire model cannot be trusted to produce accurate results. Fatal errors also disrupt the analysis for the entire system, since the system is analyzed as a simple collection of tasks. When the `fatal_error` flag is set to “yes,” code processing is halted for the current task, and the software analysis proceeds to the next task. The conditions causing the error are noted in the error file, and the final system analysis ignores any tasks that fall subject to a fatal error. It is not necessary to halt the entire system analysis because of errors experienced with an individual task, but the user must be aware that the system results produced are actually incomplete. In such cases, the user is responsible for focusing on the results of individual task analyses rather than depending upon the conclusions from the full system analysis. Certainly, some of the lower level results are useful, and it is for this reason

that errors in a single task model are not allowed to halt analysis for the rest of the system.

The final item in the parameter list for `search` is a pointer to a structure called `procedures`. This data structure holds a collection of subprogram models that are developed during the bottom-up task model construction. `procedures` is organized as a list of entries with each entry corresponding to a unique subprogram. An individual entry holds the name, filename, and code model description for a single procedure or function. Whenever code processing and model reduction are completed for a subprogram unit, the appropriate data is added to the list held in `procedures`. During the search process, every program statement is examined to see if it contains a call to one of the subprograms that has been modeled and is presently stored in `procedures`. If such a call is found, the model for the subprogram is inserted directly into the current code model. This is the method by which code modularity is recognized, exposed, and captured in the course of task model development.

As mentioned previously, `search` is essentially a grand switch statement which triggers specific responses for each of the critical constructs that might occur within a given program statement. The following paragraphs briefly describe the actions taken when dealing with the various types of critical constructs. The boldface word at the beginning of each paragraph corresponds to the name of the identifying constant used with each construct.

WFS: When a call to `wait_for_schedule` is found, a WFS entry is simply added to the current code model, provided that the `model_active` flag indicates a “yes” value. No further action is necessary.

LOOP: There are four different types of Ada program statements that could contain an instance of the word “loop.” These are listed below:

1. `end loop`
2. `while..loop`
3. `for..loop`
4. `loop (basic)`

The `search` procedure is responsible for differentiating between these possibilities and updating the current model appropriately. The test for an “end loop” statement requires a simple examination of the program statement stored in `this_line`. If an “end loop” is found, no action is taken because the presence of the word “end” will have already triggered all the necessary processes during a previous iteration of the `search` loop. The remaining three possibilities signal the beginning of a loop construct,

and it is important for `search` to identify the type of loop and its iteration limitations. For this purpose, a procedure called `process_loop` is called, and its job is to scan the program statement to determine what type of loop is being used. `process_loop` utilizes yet another procedure called `target_found` that searches the program statement in reverse to find out if a “`while`” or a “`for`” precedes the occurrence of the word “`loop`.” The results of this search determine what type of loop to add to the code model, and the value element of the loop entry is determined in one of two ways. The first method is to use the comment information preceding the loop statement. If no comment information is provided, the value element is recorded as `UNDEFINED` for `while_loops` and `for_loops` or `INFINITE` for basic `loops`, and could trigger an analysis error during the model reduction stage (Chapter 7). The second method is to search for stated limits within the program statement, and this method is used exclusively with `for_loops`. A procedure called `for_loop_found` identifies the range statement used in the loop initialization and sends it to a function called `evaluate` to determine the number of iterations. `evaluate` uses various tools to interpret the iteration range and transform it to a decimal format. If it is unsuccessful, the loop entry for the model is given a value element of `UNDEFINED`. Regardless of the loop type or its iteration limitations, the beginning of a loop construct also requires that an “`end loop`” entry is added to `end_list` to maintain an accurate picture of the code nesting.

IF: When an `if` statement is found, an appropriate entry is added to the model, provided that the `model_active` flag indicates a “`yes`” value. Recall that the nature of the test utilized by the `if` statement is irrelevant in this analysis and does not affect the development of the source code model. Also, the presence of an `if` construct requires that an “`end if`” entry is added to `end_list` to help track the source code nesting.

ELSE, ELSIF: The `else` and `elsif` statements are handled in a manner similar to the `if` statement. If the model is currently active, an appropriate entry is made and no further action is necessary.

QUEUE, RETRIEVE, SEND, RECEIVE: Whenever a valid message passing procedure call is found, an entry is added to the model, given that the model is active. An important part of this type of entry is identifying the correct size of the message being passed, and there are three possible sources for such information. The first source is the comment information provided by the programmer. If no comment information is available, the `search` procedure attempts to find the size of the message within the parameter list of the procedure call. A function called `find_parameter` extracts the size parameter from the program statement and uses the `evaluate` tool to transform the size parameter into an integer value that is returned to `search`. If either

`find_parameter` or `evaluate` is unsuccessful in determining the message size, a default value is used for the value element of the model entry. This default value is taken from the task specification file during preliminary processing and is stored in a data structure called `messages`. A pointer to `messages` is passed to `search` in the event that these default values are needed during model development.

END: As discussed previously, an `end` statement in Ada is critical to understanding the structure of the source code. When an `end` statement is found, a procedure called `end_found` attempts to match it to the last entry in `end_list`. The `end_list` entry indicates the type of construct to which the `end` statement belongs, and each case merits a unique response. Listed below are the actions taken upon finding the `end` statement for each type of framed construct:

1. **SUBPROGRAM:** First the `model_active` flag is deactivated since the `end` statement indicates that the executable code for that program unit is complete. Next the current code model stored in `skeleton` is passed to `reduce_model` (see Chapter 7) in order to refine the model and remove unnecessary entries. Finally, the subprogram name and model are added to the structure called `procedures` for use in constructing subsequent code models.
2. **PACKAGE:** Finding the end of a package is critical to following the naming rules for subprogram calls. If a procedure is defined within a package, any calls to that procedure within the package simply use the procedure name. Calls to that subprogram from outside the package must use the package name as a prefix so that the call looks like “`package_name.subprogram_name.`” In order for the code modeling process to recognize subprogram calls and associate them with the proper body of executable code, it stores subprogram names so that they always match the names that will be encountered in the current segment of source code. Thus, when the end of a package is encountered, the names of subprogram defined within that package are altered to include the package name.
3. **TASK:** When the end of a task is found, the model is deactivated, and the `finished` flag is set to a “`yes`” value in order to cease the model building process.
4. **LOOP:** If an “`end loop`” statement is found, an `END_LOOP` entry is added to the model, provided that the model is active.
5. **IF:** If an “`end if`” statement is found, an `END_IF` entry is added to the model, provided that the model is active.

6. **CASE:** If an “end case” statement is found, an END_CASE entry is added to the model, provided that the model is active.

7. **SELECT, RECORD:** No specific action is taken.

In all cases, once the tail entry in `end_list` is matched and the appropriate action is taken, that entry is removed from `end_list`. If a mismatch occurs, the `fatal_error` flag is set to “yes” to cease the current model building process.

TASK: When the word “task” is encountered, a procedure called `task_found` extracts the name of the task and compares it with the name of the task that is currently being processed. If there is a match, the `task_found` flag is set to “yes.” Also, an appropriate entry is added to `end_list` regardless of whether a match is found.

PACKAGE: The bottom-up model construction requires that all packages listed in the context clause of an application task are examined so that any executable code can be modeled. Thus, the source code processing searches for packages on an individual basis, and when the word “package” is found, a procedure called `pkg_found` compares the package name with the name of the package which it expects. If there is a match, the `pkg_found` flag is set to “yes,” and regardless of whether a match is found, an appropriate entry is added to `end_list` to signify the beginning of a package unit.

PROC: When the beginning of a subprogram body is found, a procedure called `proc_found` extracts the name of the subprogram and stores it in `procedures` along with the current filename. `proc_found` also adds an appropriate entry to `end_list` to track the source code nesting.

BEGIN: A `begin` statement is used by tasks and subprograms to indicate where the executable code starts, and the source code processing uses `begin` statements to activate the current model. If the `begin` statement belongs to the current task or any relevant subprogram, the `model_active` flag is set to “yes.”

SELECT, ACCEPT, RECORD: When an instance of “select, accept,” or “record” is found, an appropriate entry is added to `end_list`. These constructs do not affect the code model, but it is necessary to recognize their presence in order to properly track the source code nesting. It is important that `end` statements for these framed constructs are not confused with the `end` statements of other constructs.

CASE: When a `case` statement is found, an “end case” entry is added to `end_list`, and a CASE entry is added to `skeleton`, provided that the model is active. The specific object of the `case` statement is irrelevant to the modeling process and has no effect.

WHEN: Each instance of “when” that is included in the `case` statement merits its own entry in the model, for each one represents a different execution path through the source code. The specific activity within the `when` statement is modeled in the same manner as all other executable code.

procedures: The `search` procedure examines every program statement looking for subprogram calls, and the list of subprograms that have been modeled is stored in `procedures`. If a subprogram call is found and the current model is active, the subprogram model is copied directly from `procedures` into the model stored in `skeleton`. Some simple adjustments are made during the transfer to ensure that the values held in the pointer element of the model remain accurate.

In conclusion, the `search` procedure is the true workhorse of the source code processing. When `search` completes the model development for an individual task, the parsing loop terminates, and the current code model stored in `skeleton` is returned to `task_parse` for model analysis.

Chapter 7

Model Analysis

The second phase of the software analysis involves code model analysis. The preceding chapter explains how source code processing transforms Ada code into a simplified model that is stored as a collection of integers. The model analysis uses these task models produced by the source code processing to perform an efficient, automated, worst case analysis of the AFTA application task suite. All model analysis takes place independent of the original source code, and the final results are passed on to the hardware analysis stage.

In the analysis of each application task, the model analysis tools are called upon to perform two different functions. The first function is to reduce subprogram models into a more efficient form before they are stored in the `procedures` data structure. When the end of a subprogram's executable code is encountered during source code processing, the procedure called `end_found` is responsible for transferring the current subprogram model held in `skeleton` to its final storage place in `procedures`. Prior to the transfer, `end_found` sends the model to a procedure called `reduce_model`, where model development is completed and unnecessary model entries are eliminated. The methods used by `reduce_model` are fully explained in Section 7.2. The second function of the model analysis involves the final parameterization of an application task. Once `task_parse` fully develops the task model, it calls upon `find_worst_path` to identify and quantify the worst case execution path for that task. The results are recorded in the appropriate output file and stored for later use by the hardware analysis. The methods employed by `find_worst_path` are explored in Section 7.4.

Code model analysis consists of three distinct stages: model preparation, model reduction, and execution path generation. The model preparation stage takes the crude model produced by source code processing and completes its development by defining the depth and pointer elements for each model entry. The model reduction stage eliminates unnecessary model entries and produces summary entries to replace inefficient groups of model entries. Lastly, the execution path generation stage uses the model in its final form to identify all possible execution paths and compares the paths in order to single out the worst case path. The following text details the methods and motivations for each of the three stages.

7.1 Model Preparation

During source code processing, each entry is added to the current model with only the type and value elements specified. The depth, pointer, and flow elements are all left undefined during initial model construction and later added during model analysis. The model preparation stage uses the `nest_level` and `match_loops` procedures to define the depth and pointer elements for each entry of a given model, and the result is a model that is sufficiently complete for execution path analysis.

The `nest_level` procedure is responsible for defining the depth element for each model entry, and in doing so, it provides a clear picture of the nested structure of the original source code. The depth element is actually just an intermediate value that is needed later to establish the pointer element values required during execution path generation. Essentially, the depth values produced by `nest_level` can be eliminated from the model after the pointer elements are defined, but they remain part of the model for purposes of model readability and simplified debugging of the timing analysis software.

Table 7-1, Nesting Rules for `nest_level`

Entry Type	DEPTH =	nest is...
LOOP	<code>nest</code>	incremented
FOR_LOOP	<code>nest</code>	incremented
WHILE_LOOP	<code>nest</code>	incremented
IF	<code>nest</code>	incremented
ELSIF	<code>nest - 1</code>	incremented
ELSE	<code>nest - 1</code>	incremented
CASE	<code>nest</code>	incremented
WHEN	<code>nest - 1</code>	incremented
END_LOOP	<code>nest - 1</code>	decremented
END_IF	<code>nest - 1</code>	decremented
END_CASE	<code>nest - 1</code>	decremented
other	<code>nest</code>	no action taken

The `nest_level` algorithm uses an elementary loop to examine individual model entries in consecutive order. An integer called `nest` is established at an initial value of zero, and the model is examined one entry at a time from beginning to end. Certain model entries signal a deeper level of nesting in the source code while other

model entries signal the opposite. For each type of entry, `nest_level` defines the depth element according to the present value of `nest` and follows a predefined rule to determine how that entry affects the present nesting level. For instance, a `LOOP` entry signals an increase in the nesting level, and an `END_LOOP` entry signals a decrease in the nesting level. Table 7-1 summarizes the rules used in the algorithm; the second column shows what value is used to define the depth element, and the third column shows how the `nest` value changes for each entry type. Note that for any complete body of executable code such as a subprogram body or a task body, the nesting level always begins and ends at zero. If this is not the case, it signifies that a fatal error has occurred during the source code processing, and it is properly noted in the error file.

The `match_loops` procedure is responsible for linking related model entries through appropriate definitions of their pointer elements. When the model analysis attempts to identify possible execution paths through the original source code, the analysis must be able to recognize connections between related model entries in order to trace a functionally correct path. Thus the pointer element becomes a key factor during the path generation stage. The following example should clarify this concept.

```
1: if A = B then
2:   compute (X => 10);
3:   elsif B = C then
4:     compute (X => 20);
5:   else compute (X => 30);
6: end if;
```

Figure 7-1, A Sample `if` Construct

Figure 7-1 is a simple Ada `if` construct that illustrates the use of the pointer element in a source code model. There are three possible execution paths through this construct. If the conditional test in line 1 is true, execution proceeds from line 1 to line 2 to line 6. If the condition in line 1 is false and the condition in line 3 is true, the execution path includes lines 1,3,4, and 6. Lastly, if both conditionals are false, the execution path includes lines 1,3,5, and 6. Following the logic of the construct shown above seems to be a trivial task, but for an automated analysis, the logic must be built into the code model in a format that is understood by the path generation tools. When generating an execution path, the analysis must know that if the first conditional is assumed false, the execution path skips line 2 and proceeds to line 3. Likewise, if the second conditional is assumed to

be false, the execution path skips line 4 and proceeds to line 5. For this reason, the pointer elements for conditional entries in the model are needed to direct the execution path generator to the next valid section of code in the event that a particular section of code is bypassed due to the outcome of a conditional statement. The exact methods of path generation are explained in Section 7.3; at this point, it is only important to understand the motivation for the pointer element.

The `match_loops` procedure examines a given model after the depth elements are specified, and it proceeds to link related entries and install execution path logic through the pointer elements. Only certain types of model entries must have their pointer fields defined, and in such cases, the following rules apply:

1. A `LOOP` entry points to its corresponding `END_LOOP` entry.
2. A `FOR_LOOP` entry points to its corresponding `END_LOOP` entry.
3. A `WHILE_LOOP` entry points to its corresponding `END_LOOP` entry.
4. An `END_LOOP` entry points to the entry representing the beginning of the loop.
5. A `CASE` entry points to the first corresponding `WHEN` entry that follows.
6. A `WHEN` entry points to the next corresponding `WHEN` or `END_CASE` entry.
7. An `END_CASE` entry points to the corresponding `CASE` entry.
8. An `IF` entry points to the next corresponding `ELSIF`, `ELSE`, or `END_IF` entry.
9. An `ELSIF` entry points to the next corresponding `ELSIF`, `ELSE`, or `END_IF` entry.
10. An `ELSE` entry points to its corresponding `END_IF` entry.
11. An `END_IF` entry points to its corresponding `IF` entry.

Notice in the rules listed above that the key to proper pointer definition involves finding the *corresponding* entry of a particular type. This is where the depth element becomes necessary. Consider the following nested `if` construct:

```
1: if A = B then
2:   if B = C then
3:     compute(X => 10);
4:   end if;
5:   else compute(X => 20);
6: end if;
```

Figure 7-2, A Sample Nested `if` Construct

When setting up pointer elements for the construct shown in Figure 7-2, there is some confusion if the depth element is not utilized. The pointer definition rules state that the IF entry taken from line 1 should point to its *corresponding* END_IF entry, which is taken from line 6 in this example. If the `match_loops` procedure defines the pointer element in accordance with the *next* END_IF entry, it commits an error by designating the END_IF entry taken from line 4 of the preceding figure. The depth element is the key to distinguishing between the “end if” statements in lines 4 and 6. Notice that for both the outer if construct and the nested if construct, the related “if” and “end if” statements have identical depth values defined within the model. Thus, for an IF entry, `match_loops` is able to find the corresponding END_IF entry by searching for the next END_IF entry that has a depth value identical to that of the IF entry. Essentially, the depth element allows the automated model analysis tools to understand the concept of corresponding model entries.

7.2 Model Reduction

After a given model is processed by `nest_level` and `match_loops`, that model is complete, but it is not necessarily an efficient representation of the original source code. There are situations where model entries can be eliminated without altering the analysis results, and there are other situations where a group of model entries can be summarized effectively by a single model entry in order to reduce the size and complexity of the model.

The process of entry elimination is reserved for situations involving empty loops. An empty loop is defined as a LOOP entry followed directly by an END_LOOP entry. It is reasonable to assume that the original source code does not contain such empty loops, but it is incorrect to make the same assumption about the source code model. The concept of abstraction allows the code model to omit details that are considered irrelevant to the timing analysis, and it is possible that the source code processing could add a loop structure to the model and simultaneously find nothing within that loop that is important enough to record in the model. The result is an empty loop that adds unnecessary complexity to the task of generating execution paths for the complete model, and the proper course of action is simply to delete the loop from the model. However, there is a restriction on the types of empty loops that can be eliminated without altering the analysis results. The primary constraint is that the loop’s iteration limit must be defined before the loop can be deleted. In other words, if the value element for the LOOP entry is listed as UNDEFINED or INFINITE, the loop must remain in the model to

ensure that the undefined behavior of the construct is properly noted in the error file during future analysis. As long as the loop's value element is defined, any empty `for...loop`, `while...loop`, or basic `loop` is removed during model reduction. This elimination process may seem to be a reckless omission of detail by the model analysis, but the responsibility for model accuracy actually lies with the source code processing. If the source code processing recognizes a program statement as significant to the timing analysis, it is recorded in the model and accounted for in the model analysis. If a program statement is ignored in the model development and an empty loop results, the model analysis is held accountable for it.

When the model analysis attempts to generate execution paths, the processing workload is significantly decreased by reducing the model to a more efficient format. Consider the following loop and its model:

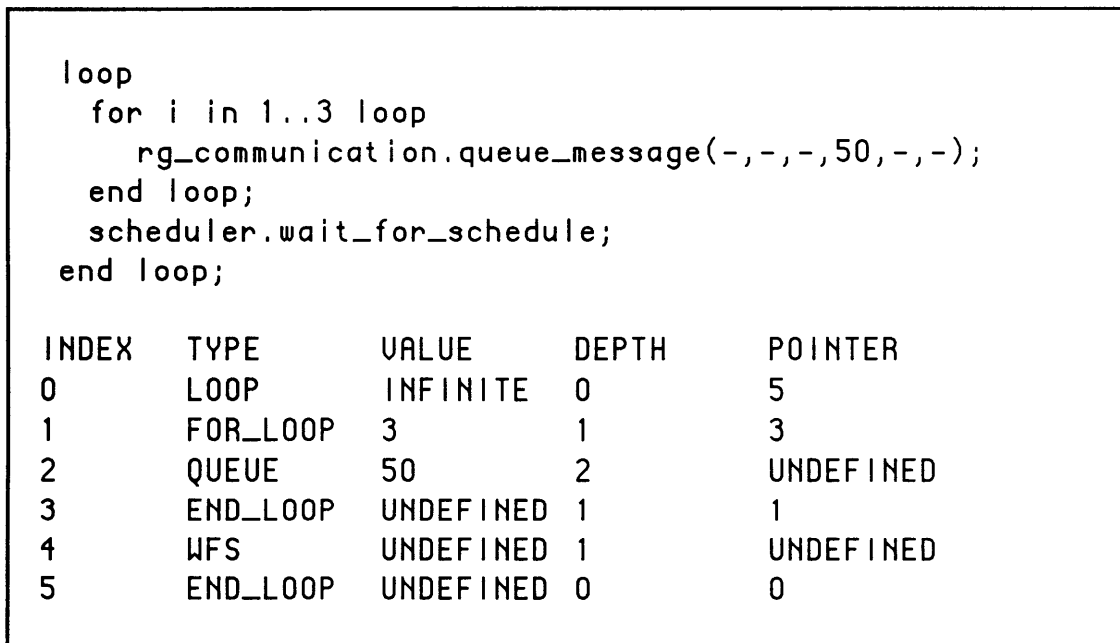


Figure 7-3, A Sample Nested Loop and its Model

When the model analysis generates an execution path for the model shown in Figure 7-3, it produces the following sequence: 0,1,2,3,1,2,3,1,2,3,4. Notice the repetition of the inner loop, and imagine what the sequence would look like if the loop bound was 100 iterations rather than three iterations. In situations such as this, the path generation process is inefficient and at times unmanageable. It is therefore advantageous to summarize the inner loop with a single model entry called a `COUNTER_SET` entry. The `COUNTER_SET` entry parameterizes the execution information held in a contiguous

group of model entries and is then used to replace that group of entries. The COUNTER_SET entry is merely a collection of integers that indicate how many times each critical construct is encountered during execution of a given segment of code; its format is shown below:

counter_set	(structure)
num_sent	(integer)
num_read	(integer)
num_queued	(integer)
num_retrieved	(integer)
rg_log_entries	(integer)
debug_entries	(integer)
io_utils	(integer)

Figure 7-4, Format for a COUNTER_SET Entry

Notice that the COUNTER_SET structure includes a counter for every critical construct that does not pertain to the control of execution flow. In other words, it accounts for events like operating system calls while ignoring the elements of loop constructs and conditional statements. This structure is also used to store the final parameterized representation of the application task that is ultimately produced by the model analysis. In this particular example, the COUNTER_SET entry records a value of 150 bytes for the num_queued counter because a 50 byte message is queued for each of the three inner loop iterations. All other counters remain at their initial values of zero. In this manner, COUNTER_SET records all the necessary execution information held by entries 1,2, and 3 in the model of Figure 7-3, and it greatly simplifies the process of execution path generation. Refer to Figure 7-5 for an updated version of the model shown in Figure 7-3.

INDEX	TYPE	VALUE	DEPTH	POINTER
0	LOOP	INFINITE	0	3
1	COUNTER_SET	0	1	UNDEFINED
2	WFS	UNDEFINED	1	UNDEFINED
3	END_LOOP	UNDEFINED	0	0

Figure 7-5, An Updated Model

Notice that the inner loop disappears, and the pointer value for the top entry is adjusted accordingly. The `COUNTER_SET` entry is given a value of 0 since it is the first entry of its type in the model, and it assumes a depth value equal to the depths of the first and last entries in the set of entries that is replaced. Part of the data structure used to store code models is reserved for an array of `COUNTER_SET` records having the format shown in Figure 7-4. The value element of the `COUNTER_SET` entry corresponds to the `COUNTER_SET` array index used to access the counters. The proper execution path sequence for the updated model is now shortened to the following: 0,1, and 2. The method by which a `COUNTER_SET` entry is accounted for in the quantification and evaluation of execution paths is explored in Section 7.3.

The process of summarizing groups of model entries may seem to be modeling overkill, but it is necessary to make the task of execution path generation a manageable one. It also may seem that the application of this process is not properly bounded because it theoretically could be applied to any coherent set of entries, but this is not so. This type of model reduction is strictly limited to the following classes of constructs:

1. A `for . . loop` with a defined iteration limit and no WFS entry within the loop.
2. A `while . . loop` with a defined iteration limit and no WFS entry within the loop.
3. A basic `loop` with a defined iteration limit and no WFS entry within the loop.
4. An `if` construct containing no WFS entry.
5. A `case` construct containing no WFS entry.

The reason why these constructs must not encapsulate a call to `wait_for_schedule` is that the WFS call is critical to the process of execution path generation. Recall that a single execution cycle consists of all program statements executed between two consecutive WFS calls. As such, all WFS calls must remain in the model for any subprogram or application task so that they can be referenced as beginning and end points for various execution paths. Concealing a WFS call inside the summary of a `COUNTER_SET` entry destroys the functionality of WFS entries within the realm of model analysis.

The transformation from a qualified group of entries to a single `COUNTER_SET` entry takes place in a procedure called `crunch`. This procedure is called five separate times by its parent procedure, `reduce_model`, and each call corresponds to one of the five types of constructs that are listed above. Once invoked, `crunch` first searches for qualified groups of entries belonging to a particular type of construct; for instance, it may search for all `while . . loops` that contain no WFS entries. For each group that is

found, `crunch` creates a temporary model containing only the entries from that group, and it sends the temporary model to a procedure called `generate_paths` for a “mode 1” path analysis. The function of `generate_paths` is to identify all execution paths through a given model, parameterize them, compare them, and single out the worst path. A “mode 2” path analysis refers to the process of identifying and comparing execution paths that proceed from one WFS call to the next WFS call. This is the type of analysis that is performed on the complete application task model in search of the worst case parameterization of that task. A “mode 1” analysis involves finding all execution paths that strictly proceed from the first model entry to the last model entry. Since `crunch` deals only with groups of model entries containing no WFS calls, a “mode 1” analysis is the correct and logical choice. It may seem too complex to give the same procedure the responsibility for performing two different types of analyses, but the implementations of “mode 1” and “mode 2” path analyses are so similar that using two versions of `generate_paths` would create unnecessary redundancy. The results produced by `generate_paths` are sent back to `crunch` in the form of a `COUNTER_SET` entry, and `crunch` performs the `COUNTER_SET` substitution within the original subprogram model or task model.

The example shown in Figure 7-3 illustrates how the length of execution path sequences is significantly reduced through model reduction. It is also important to understand how model reduction further decreases the analysis workload by eliminating possible execution paths from consideration during the final task analysis. Suppose a task model contains a basic `if` construct that includes both an `elsif` and an `else` statement. Such a construct has two conditional statements and three possible paths of execution. The `crunch` procedure finds the construct, develops an independent model from it, and submits it to `generate_paths` for a “mode 1” analysis. The path analysis generates the three possible paths through the construct, parameterizes them, compares them, and singles out the worst of the three. It returns its results to `crunch` in the form of parameter set, and `crunch` replaces the original `if` construct with a `COUNTER_SET` entry and reduces the size and complexity of the task model. The actual model reduction is a positive result from this process, but it is not the most significant result in this particular case. It is more important to recognize the fact that an `if` construct that allows three distinct paths is replaced by a `COUNTER_SET` entry that allows only one path. Thus the total number of possible execution paths through the task model is decreased by a *factor* of three. By isolating the `if` construct to identify its own worst case path, the workload required to process the full task model is tremendously reduced. In a similar

manner, replacing a `case` construct with four options cuts the number of execution paths through a task by a factor of four.

7.3 Execution Path Generation

Recall that the purpose of source code modeling is to provide an effective and efficient means to identify the worst case execution path through a given application task. The processes of model development, preparation, and reduction lay the foundation for the most crucial stage of the analysis: execution path generation. The path analysis process has the following three objectives:

1. To identify all possible execution paths through a given model.
2. To accurately parameterize each path generated.
3. To evaluate the parameterizations and identify a single worst case path.

In order to understand the methods employed by the path analysis, it is best to first explore the general nature of the task at hand.

```
if A = B then compute (X => 10);
  elsif A = C then
    if B = D then compute (X => 20)
      elsif B = 10 then compute (X => 25)
        else compute (X => 30)
    end if;
  elsif A = D then compute (X => 35);
  else A = 10;
end if;
```

Figure 7-6, A Nested `if` Construct

Consider the nested `if` construct shown in Figure 7-6. There are six possible paths through this construct, and each path must be explored and compared against the others with respect to execution delay. A decision tree developed from this construct is shown in Figure 7-7; notice that the circled items represent the leaves of the tree, which are actually the endpoints of the six execution paths. Each conditional statement in the construct is shown as a decision point in the tree, and each decision has two possible results, as represented by the left and right branches. The various locations on the tree are uniquely expressed in a binary format based upon which branch is taken at each decision point encountered along the path to that location. A left branch is represented by a '1,' and a right branch is represented by a '0.' The digits are listed in the order in which the

decisions are made. The critical result of all these conventions is that the location of each leaf or endpoint is expressed as a binary number that directly correlates to the execution path taken to reach that endpoint.

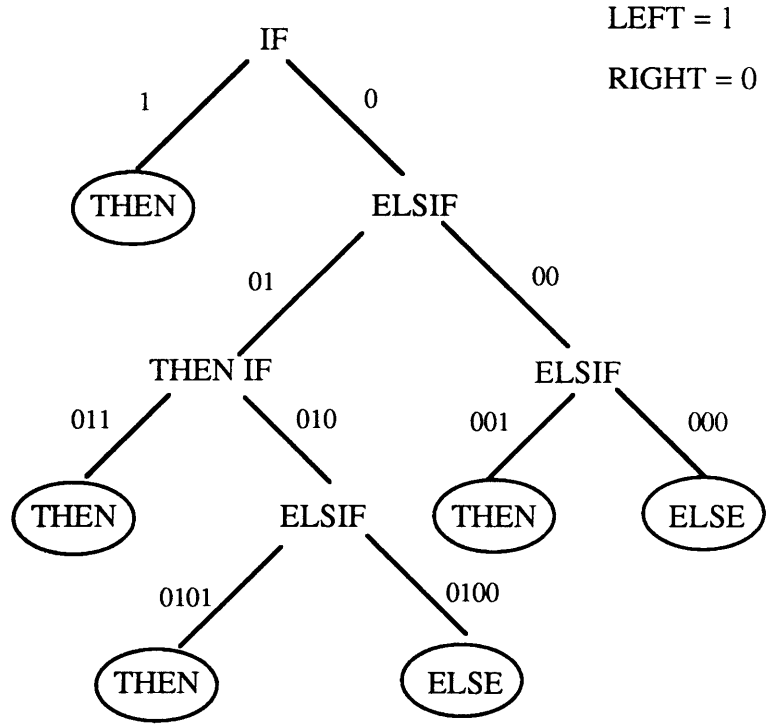


Figure 7-7, A Nested if Decision Tree

A manual path analysis would probably utilize a decision tree similar to that shown in Figure 7-7, for once the tree is developed, it is a trivial task to map out all the possible execution paths. The challenge of the automated model analysis is to develop an algorithm that uses analogous methods to produce the same results as the manual analysis. The resulting algorithm must achieve the following three goals:

1. The algorithm must be able to comprehend the functionality of the code model and generate execution paths using the tree-like format shown in the figure above.
2. It must be able to fully explore the paths through the tree and know when all paths have been explored.
3. The algorithm must be efficient enough to avoid tracing the same execution path more than once.

The predefined format of the code model and the ability to define a path through the tree as a binary number are key facilitating factors in the development of the path generation

algorithm. The following explanation and walk-through example highlights the methods used to achieve the goals listed above.

The path generation algorithm primarily focuses upon the process of generating the binary values that correspond to the individual paths and their respective endpoints. Two integer variables control all activity involved; one is called `shift_number` and the other is known as the `decision` integer. `shift_number` tracks the length of the current execution path, and the value of the `decision` integer predetermines the decisions made at each decision point along the current path. It is best to think of the `decision` integer in terms of its binary representation, for the '1's and '0's are what actually determine the decisions made. `decision` is initialized to a value of zero, which determines the first path that is explored, and after each path is completed, the value of `decision` is updated in such a manner that it predetermines the next path to be examined. The updating process is designed specifically to ensure that all paths are explored once and only once and to notify the algorithm when the exploration is complete. For an illustrative example, the path generation algorithm is applied to the tree in Figure 7-7, and the values of `shift_number` and `decision` at each step of the algorithm are laid out in Table 7-2.

The `decision` integer is shown in an eight bit binary format with the most significant bit separated from the others. The MSB is called the "done bit" because when it changes from a '0' to a '1,' it signals the algorithm that the path generation process is complete. The presence of the "done bit" leaves only 7 bits of the `decision` integer for use in the path generation process, and this is signified in the initial value of `shift_number`. The algorithm begins with `decision` initialized to zero, and it starts at the top of the tree with the first decision point. The outcome of the first decision is determined by the value of the bit adjacent to the "done bit," and accordingly the right branch is chosen. Once a decision bit is referenced, it is not used again until the next path is explored. Subsequent decisions are made by referencing the next least significant bit within the decision integer. For every decision bit used, `shift_number` is decremented once to track the number of available decision bits remaining. For the first execution path, all decision bits are '0's, and the right branch is taken at three successive decision points until the endpoint labeled "000" is reached. As expected, the binary value of the endpoint is identical to the series of decision bits referenced in arriving at that location. The second line of the table shows the three decision bits used, and it lists `shift_number` with a value of 4 since there are four decision bits unused for the first path. At this point, the decision integer is updated to prepare for the next path. The update consists of adding a '1' to the last decision bit used, and this is accomplished by taking an integer value of 1

and shifting its bits left by 4 places. Notice the correspondence to the current value of `shift_number`.

Table 7-2, A Path Generation Example

Path	Decision Integer	Shift_Number	Comment
1	0 0000000	7	Initial value of decision
1	0 000 _ _ _ _	4	First three decision bits are used
2	0 0010000	7	A 1 has been added to last bit used
2	0 001 _ _ _ _	4	First three decision bits are used
3	0 0100000	7	A 1 has been added to last bit used
3	0 0100 _ _ _	3	First four decision bits are used
4	0 0101000	7	Add 1 to the last active bit
4	0 0101 _ _ _	3	First four decision bits are used
5	0 0110000	7	Add 1 to the last active bit
5	0 011 _ _ _ _	4	Only three decision bits needed
6	0 1000000	7	Add 1 to the last active bit
6	0 1 _ _ _ _ _	6	Only a single decision bit is needed
7	1 0000000	7	The MSB changes; process is done

The updated value of `decision` predetermines the decisions that will be made in exploring the next new path. It guarantees that the new path will not be a repetition of any previously explored path because the decision integer assumes a new and unique value. The next state of the decision integer always depends upon the present state, and the use of addition to transition between states guarantees that all states are unique. The only way that a state can be repeated is if the integer becomes so large that its value rolls over, but before this happens, the “done bit” is forced to transition from a ‘0’ to a ‘1.’ Such a transition automatically triggers the end of the algorithm and thus prevents state repetition. The basic idea behind the update methodology is that changing the last decision made leads to at least one new endpoint, and it possibly leads to a whole series of new endpoints springing from previously unencountered decision points. If the last decision that was made corresponds to a left branch decision (a decision bit equal to ‘1’), adding a ‘1’ to the appropriate bit invokes a “bit carry” and effectively changes a decision bit further upstream. It simultaneously clears or refreshes all the downstream decision bits. The refresh activity ensures a complete exploration of any new branch in the same manner that the initial zero value for `decision` ensures complete exploration of the

entire tree. The update process is deceptively simple, for a seemingly trivial addition process automatically deals with all issues of path repetition, path exhaustion, and notification of completion. The third line of the table lists the updated `decision` integer that is used to generate the second path. As with the first path, the first three decision bits are used, and this leads to a bit carry when updating `decision` for the third path. The bit carry appropriately changes the value of the second decision bit and refreshes the value of any downstream decision bits. The algorithm continues as described until all six paths are explored. While updating the decision integer in preparation for a seventh path, the “done bit” registers a ‘1,’ and the algorithm automatically terminates.

The algorithm implemented in `generate_paths` is quite similar to the one described in the preceding example, but it is slightly more complex because it must transform the code model into a decision tree format as it executes the path generation algorithm. A flow chart for `generate_paths` is included on the following page. As described previously, this procedure performs two types of path analyses. The algorithm is identical for both types; the main differences between the two are the starting points and the ending conditions for individual paths. The “mode 2” analysis is for finding paths through the complete task model. Each path begins at the entry following a WFS entry and ends with a WFS entry. The “mode 1” analysis is used during model reduction, and every path begins at the first model entry and ends with the last model entry. As the procedure proceeds through the path generation, the index values for entries encountered along the path are stored in a one-dimensional integer array called `path`. When called, `generate_paths` is given the index value for the first entry on the path. It evaluates that entry to determine if it is a decision point, and if it is, a procedure called `decide` is used to manipulate the `decision` integer to find the next step along the path. Note that the `decision` integer is expanded to a 64 bit number in this implementation of the algorithm to account for the complexity of complete task models. Only a few types of model entries prompt a decision process; these are listed below:

1. IF
2. ELSIF
3. ELSE
4. WHEN
5. WHILE_LOOP
6. END_LOOP

Depending upon the outcome of the decision, the flow element is sometimes activated for certain entries that follow the decision point. Consider the case of an `if` construct in

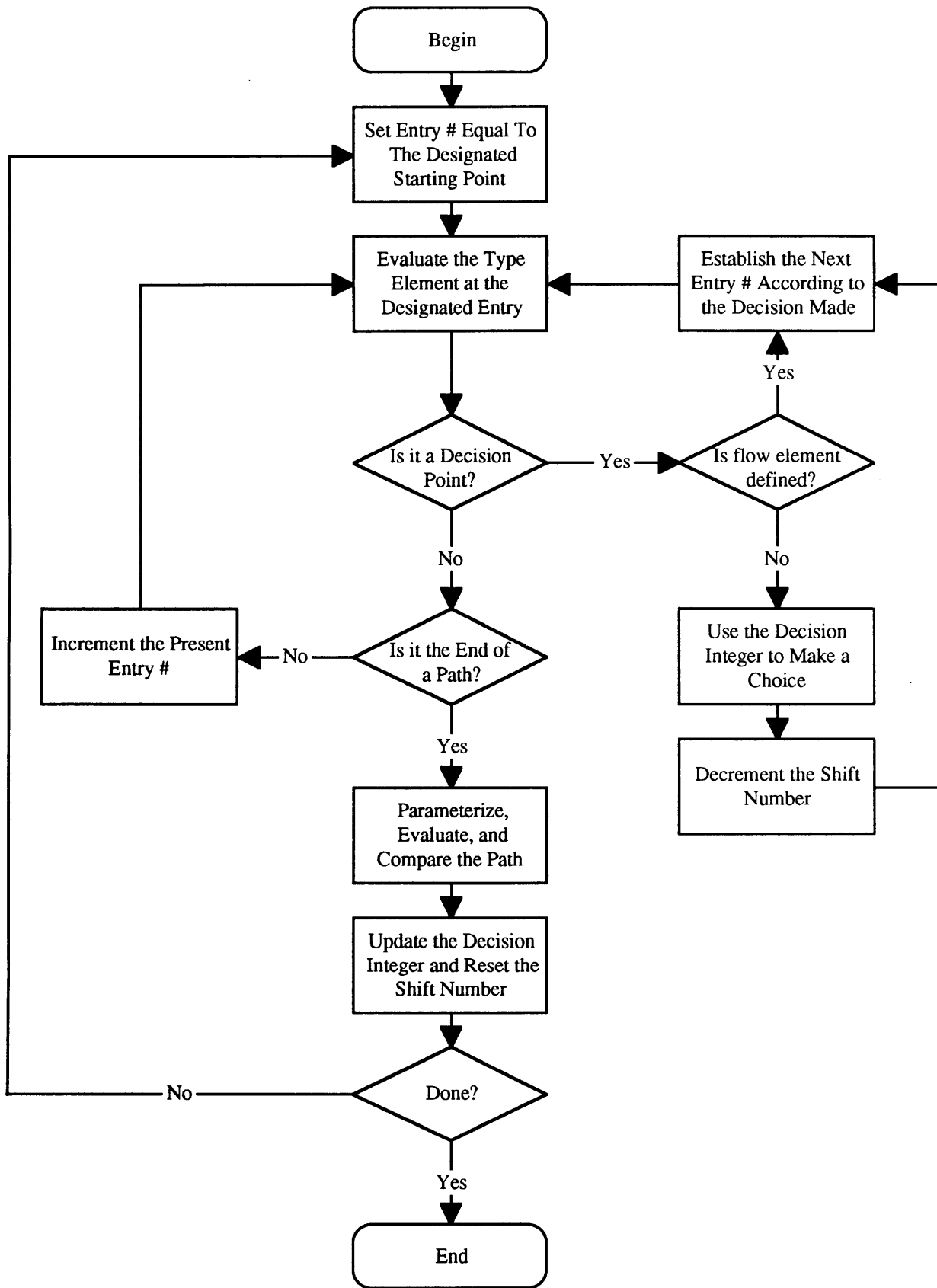


Figure 7-8, The generate_paths Algorithm

which the `if` condition is assumed to be true. This corresponds to taking the first left branch in the decision tree of Figure 7-7. Once the left branch is chosen, all `elsif` and `else` branches within that `if` construct are excluded from the execution path. The flow elements of the ELSIF and ELSE branch entries are set to a value called `NO_EXEC` in order to signal the algorithm that those entries are no longer valid steps along the path. When `generate_paths` encounters a `NO_EXEC` entry within its path, it knows not to add the index value of that entry to the current `path` array, and it proceeds to examine the next consecutive entry.

Now consider the situation where an `if` condition is assumed to be false, and the `if..then` branch is excluded from the path. The value of the pointer element at the IF entry allows `generate_paths` to skip the entries of the `if..then` branch and go directly to the next decision point or to the conclusion of the `if` construct, whichever comes first. When `generate_paths` encounters a model entry that is not a decision point and does not have its flow entry defined, it merely adds that entry index to the path and continues on to the next consecutive entry.

When a path trace is complete, `generate_paths` sends the `path` array to a procedure called `parameterize`, which quantifies the path in terms of the `counter_set` structure described in Section 4.6.2. The parameterization essentially retraces the path and examines the type and value elements of the appropriate model entries. When it finds an entry which contributes to one of the execution delay factors specified in the `counter_set` structure, it simply updates the value of the related counter according to the value element of the model entry. The result of this process is a simple collection of counters that quantify the timing behavior of a given path through a given model. It may seem that the `counter_set` structure is too limited in scope to properly characterize the timing behavior of an execution path, but keep in mind that the parameter list can and will be expanded in future efforts to improve the code modeling process. The current set of critical constructs that drive the code model development will grow as the AFTA testing and system evaluation progress.

Once the parameterization is complete, the parameter set is returned to `generate_paths`, which sends it to the hardware analysis tools where an actual lower bound on execution delay is estimated based upon the parameter values submitted. The hardware analysis methods are explored in Chapter 8. Given a single delay value for each path, `generate_paths` easily compares the paths and identifies which path causes the greatest execution delay. After all paths have been traced, parameterized, evaluated, and compared, `generate_paths` returns to its parent procedure a single set

of parameters that represent the worst case path through the given model for the given mode of analysis.

7.4 Managing Model Analysis

The preceding sections focus upon the motivations and methodology of code model analysis. It is now useful to discuss the manner in which these analysis activities are managed.

A procedure called `reduce_model` is a simple series of procedure calls that oversees model preparation and reduction. It begins with a call to `nest_level`, which is followed by a call to `match_loops`. Once the model preparation is complete, the model is submitted to a procedure called `model_ok`, where the model's validity is confirmed prior to any further processing. `model_ok` performs a series of routine checks to ensure that the model is able to undergo execution path analysis without causing a run-time error for the timing analysis software. A list of these diagnostics is included below:

1. Does model depth begin and end at zero?
2. Are all `loop`, `if`, and `case` constructs complete with `end` statements?
3. Are there any true infinite loops with no WFS calls inside?
4. If the model is a task model, is there at least one WFS entry?
5. Are any non-existent `COUNTER_SET` entries included in the model?

Any errors identified by `model_ok` are specifically noted in the error file. Obviously, no errors are expected, but this type of preventive check helps avoid a situation in which a single erroneous subprogram model crashes the entire AFTA timing analysis. After the model is examined, `reduce_model` proceeds to a series of five calls to the `crunch` procedure. Recall that `crunch` performs model reduction according to a specified type of Ada construct. This procedure is called once for each of the following constructs: `loops` (basic), `for...loops`, `while...loops`, `if` constructs, and `case` constructs. After model reduction is complete, `reduce_model` terminates and returns the final model to its parent procedure.

The `reduce_model` procedure is called in two types of situations. First, it is called each time the source code processing completes a subprogram model, and it is invoked from the procedure called `end_found`. The subprogram model returned by `reduce_model` is stored in the procedures data structure for later reference. The second situation involves model analysis for a single application task. After an application task model is constructed, `task_parse` submits it to a procedure called `find_worst_path`, which is responsible for reducing a task model to a single

parameter set representing the worst case execution path. `find_worst_path` begins by calling upon `reduce_model` to perform model preparation and reduction. It then examines the resulting model to find all WFS entries. Since a single cycle execution path must begin and end at a call to `wait_for_schedule`, `find_worst_path` uses the list of WFS entries as starting points for “mode 2” path analyses. For each WFS entry, `find_worst_path` calls upon `generate_paths` to perform a “mode 2” analysis using that entry as the starting point. `generate_paths` returns the parameter set representing the worst case path beginning from the specified WFS entry, and `find_worst_path` compares the parameter sets returned for all WFS entries. The result is a single worst case parameter set that is used to represent the entire application task for the remainder of the timing analysis. `find_worst_path` returns the parameter set to its parent procedure, `task_parse`, and the software analysis for that application task is then complete.

Chapter 8

Hardware Model Analysis

8.1 Introduction

The hardware model analysis is the last of the three phases of the AFTA timing analysis. The fundamental output of the first two phases (preliminary processing and software analysis) is a collection of worst case delay parameterizations -- one for each task instantiation in the task suite. The primary function of the hardware analysis is to examine these parameterizations with reference to the AFTA system configuration in order to predict performance failures. In this role, the hardware analysis is essentially an integration phase, for it utilizes models of the rate group scheduling system and the AFTA virtual group configuration in performing a comprehensive examination of the task models. During the software analysis, task models are developed and analyzed strictly on an individual basis in an effort to quantify a single worst case execution delay for each task instantiation. In contrast, the hardware analysis views tasks collectively according to their virtual group designations, and it uses known performance characteristics of the operating system to determine if, under worst case conditions, the task groupings as a whole can satisfy the hard-real-time constraints of the rate group scheduling system. A discussion of this process begins with Section 8.4.

A second function of the hardware analysis involves calculation of a lower bound delay value for a given execution path. The results of this calculation are needed by the software analysis to effectively compare different execution paths through the same code model and thereby identify the worst case path through that model. This function is so closely associated with the software analysis that it is difficult to distinctly classify it as part of the hardware analysis, but since it depends heavily upon knowledge of system specific delays, it is best to discuss this calculation process in conjunction with other hardware-related analyses. This process is presented in Section 8.3.

8.2 Benchmarking

The timing behavior of a hard-real-time system is critical to its success, and a proper understanding of the timing behavior enables the user to maximize a system's performance without compromising its effectiveness. To this end, Draper Lab is conducting a thorough study of the AFTA's performance characteristics, and the results to date are reported in [CLAS93]. The objectives of the performance measurement study are as follows:

1. To develop analytical models useful in predicting system performance under various configurations and workloads [CLAS93].
2. To be able to quantify system overhead on a frame by frame basis in order to calculate the time available for application tasks [CLAS93].
3. To identify potential performance bottlenecks so that they are eliminated in a cost-effective manner at an early stage of development [CLAS93].

The first two objectives listed above are closely related to the goals of the AFTA timing analysis. The execution model that has emerged from the performance measurement study is used by the hardware analysis to examine the timing behavior of virtual groups within the confines of rate group scheduling. Also, the actual performance measurement data enables the timing analysis to quantify both overhead and application task delays for a particular virtual group on a frame by frame basis. It is these calculations that lead to the final performance failure predictions.

Figure 8-1 shows a model of the operating system overhead involved in each minor frame. This figure is not drawn to scale and is intended to illustrate the chronological progression of overhead tasks relative to scheduled interrupts.

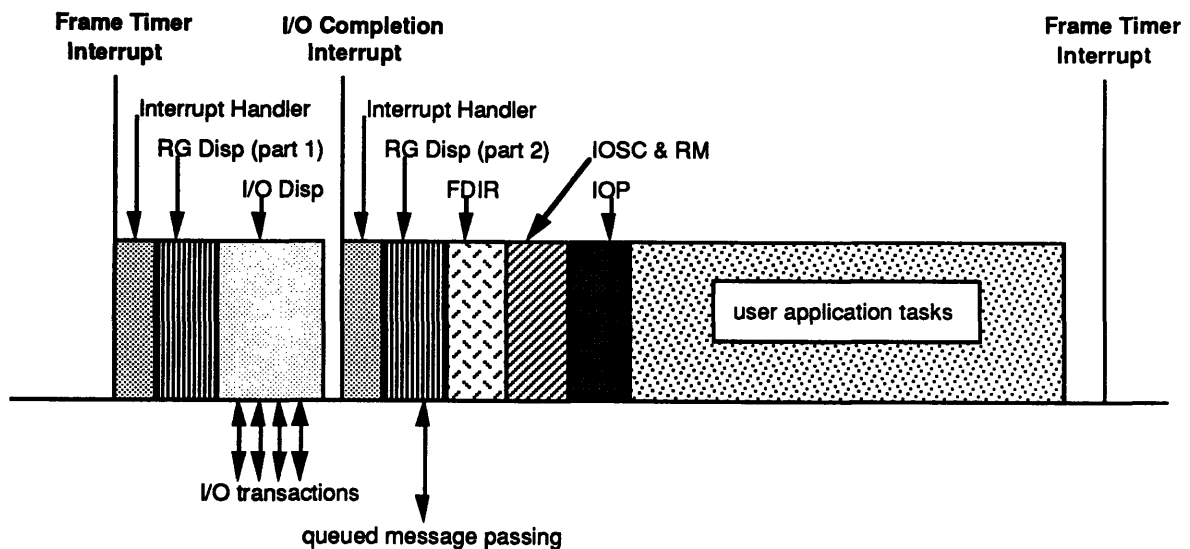


Figure 8-1, Minor Frame Overhead Model [CLAS93]

Each minor frame begins with a hardware-generated timer interrupt that is scheduled by the operating system, and the frame time is partitioned into two sections by a second timer interrupt known as the I/O completion interrupt. The first portion of the frame is

dedicated to operating system functions and I/O operations, and the second portion is filled with another round of operating system functions followed by user application task time. Notice that the amount of processing time allotted to the application task suite is clearly affected by the amount of time dedicated to operating system overhead within the second portion of the frame. The software analysis determines how much time an application task requires under worst case conditions; the hardware analysis then groups tasks according to their designated VGs and further calculates worst case operating system overhead for each VG within each minor frame. The total of all required processing times for a particular virtual group is then compared against the time allotted for the second portion of the minor frame to determine if a performance failure could occur. In simple terms, a performance failure indicates that the second portion of one minor frame overlaps into the first portion of the next minor frame. This type of “overrun check” is the main thrust of the hardware phase of the AFTA timing analysis, but it is important to notice that a second type of overrun condition can also occur. If the operating system overhead within the first portion of the frame exceeds its allotted time, it overlaps into the second portion of the frame. This type of overrun is heavily dependent upon the activity of the I/O dispatcher, but at this point in the AFTA’s development, the implementation of I/O operations is not well-defined. As such, the current timing analysis does not attempt to quantify the I/O dispatcher execution delay or predict this form of intraframe overrun.

The following paragraphs give a brief description of the various overhead tasks that are executed as part of each minor frame. A simple equation is included with each task description to indicate how much frame time the task occupies and what variables affect its execution delay. These equations are taken from [CLAS93], and they are based upon data taken during AFTA performance measurement study.

IH₁: The interrupt handler updates the current time value held by each member of the virtual group. For fault-tolerant operation, it is critical that all VG members maintain the same clock value, and this time update serves to eliminate clock skew. The interrupt handler also schedules the next interrupt time, which in this case is the I/O completion interrupt. The most time consuming portion of the interrupt handler is the message scoop. This involves the transfer of all message packets destined for the VG from the network element’s dual port RAM to the local memory space of the individual members of the VG. The delay incurred by the scoop process varies according to the number of packets added to the virtual group’s network element input buffers since the last scoop.

$$IH_1 = (110 * \text{number_of_packets}) + 103 (\mu\text{sec})$$

RGD₁: This is the first of two manifestations of the rate group dispatcher. Its first function is to establish a constant time reference for each of the various rate groups whose frame boundaries coincide with the beginning of the current minor frame. This reference is set to the congruent clock value established by the interrupt handler at the beginning of the frame, and it ensures that all members of a VG use identical values for any time-based calculations executed within the application tasks. For the present rate group designations, the RG4 reference value is updated every minor frame, whereas for RG1, this value is updated only once every eight minor frames. Refer to Figure 8-2 (page 98) for an illustration of the rate group frame organization and frame boundaries. The second function of the rate group dispatcher is to check for overrun conditions. It first looks for an overrun on the most recent execution of the rate group dispatcher, and then it checks for overruns on any tasks that were scheduled to complete within the previous minor frame. The overrun checking occupies the majority of the time for the RGD, and its delay is a function of the number of application tasks that were supposed to complete within the last minor frame. The longest delay is incurred in minor frame 0 (refer to Figure 8-2) because tasks in all four rate groups are scheduled to complete in the preceding minor frame (#7). In contrast, the shortest delay is incurred in minor frames 1,3,5, and 7 since the RGD only monitors RG4 tasks within those minor frames.

$$RGD_1 = (10 * \textit{number_of_suspended_tasks}) + 69 \text{ (\mu sec)}$$

IOD: The I/O dispatcher is responsible for handling all I/O requests. It begins by initiating all requested outgoing data transmissions and then waits for a predetermined time period to allow for completed transmission of all data. Following this idle period, the IOD begins reading in any received data, and the duration of this task is a function of the number of input requests and the total amount of data transmitted. At this stage of the AFTA design, the I/O operations are not fully refined, and the delay incurred by the IOD has not been accurately measured or analyzed. As such, the AFTA timing analysis presently overlooks all execution delay incurred by I/O operations and related operating system overhead. It will be a feasible task to later include such considerations in future revisions of the timing analysis.

IH₂: The execution of the interrupt handler in the second portion of the minor frame is identical to that which occurs in the first portion of the frame. This second execution serves to scoop all message packets delivered for the VG since the previous

execution of the interrupt handler. The use of two scoop operations within the same minor frame is needed to maintain synchronization between members of the same VG.

$$IH_2 = (110 * \text{number_of_packets}) + 103 (\mu\text{sec})$$

RGD₂: The second part of the rate group dispatcher is responsible for a number of functions. It begins by checking for overrun conditions on the first part of the RGD and on the preceding execution of the I/O dispatcher. This is followed by the `send_queue` and `update_queue` functions. Together these functions perform a data transfer of all message packets that were queued by tasks which completed their execution cycles during the previous minor frame. The delay incurred by these functions therefore varies according to the number of tasks whose messages require transfer as well as the size of those messages. A summary equation is given below:

$$\text{Send_and_Update_Queue (per task)} = (123 * \text{number_of_packets}) - 12 (\mu\text{sec})$$

Following the message passing operations, the RGD schedules all rate group tasks whose frame boundary coincides with the beginning of the current minor frame. For example, during minor frame 0, the RGD schedules all rate group tasks since all rate group frames begin anew with minor frame 0. In contrast, during minor frames 1,3,5, and 7, the RGD only schedules RG4 tasks because no other rate group frames begin with these minor frames. A summary of the delay incurred by task scheduling is shown below:

$$\text{Schedule_Tasks (per Rate Group)} = (26 * \text{number_of_rg_tasks}) + 15 (\mu\text{sec})$$

By combining the two previous equations and adding a constant delay incurred by the overrun checking and other minor functions, the total delay for the second part of the rate group dispatcher is summarized as follows:

$$RGD_2 = \sum_{i=1}^{num_tsk} [(87 * num_pkt_i) + 27] + \sum_{i=1}^{num_rg_tsk} [26 * num_rg_tsk_i + 15] + 49 (\mu\text{sec})$$

- where: *num_tsk* is the number of tasks that completed execution cycles during the previous minor frame.
num_pkt_i is the number of packets that task *i* queued during its previous execution cycle.
num_rg_tsk is the number of rate groups whose frame boundary coincides with the beginning of the current minor frame.
num_rg_tsk_i is the number of tasks belonging to rate group *i*

FDIR: The fault detection identification and recovery task is the software complement to the AFTA's hardware redundancy and fault masking capabilities. Local FDIR enables a virtual group to monitor itself and potentially perform some recovery

operation. It is executed on all virtual groups and its execution delay is a known constant of 84 μ sec. System FDIR allows the AFTA to monitor the global system and to determine the health of shared components such as the network elements [AFTA91]. It executes as an RG4 task on a single redundant VG known as the system VG. In its present developmental state, system FDIR incurs a 1316 μ sec delay.

IOSC: The I/O source congruency manager ensures that all members of a virtual group receive identical copies of any input value read by one or more members of the group. This involves complete message passing operations, and the delay incurred is a function of the number of I/O input values as well as the number of VG members receiving these values from external systems (most likely one member or all members). Since the AFTA's I/O capabilities are not yet refined and ready for testing, there is no data on this source of overhead.

IOP: The I/O processing task resolves multiple input values into a single quantity that is used by all members of the VG. Suppose that each member of a triplex VG is interfaced with an external air temperature sensor. It is not likely that all three sensors would return identical values to their respective processors, and it is therefore necessary for the IOP to implement some algorithm for data resolution. The IOP is not fully implemented at this time, but initial measurements show that the minimal IOP overhead is 15 μ sec.

In addition to overhead evaluation, the performance measurement study also focuses on operating system functions related to user application tasks. For example, data has been taken to determine the amount of delay required for a context switch, which occurs every time a task completes an execution cycle and suspends itself so that the next scheduled task can begin execution. Results indicate that the average context switch requires 19 μ sec. Naturally, the amount of time devoted to context switching varies according to the number of tasks that complete execution during a given minor frame. For the sake of this analysis, it is assumed that all rate group tasks complete execution during the last minor frame of their respective rate group frames. For instance, it is assumed that RG1 tasks complete execution during minor frame 7, and RG2 tasks complete their cycles during minor frames 3 and 7.

The most time consuming operating system calls utilized by application tasks are the message passing functions. Evaluations of these operations yield the following results:

$$\begin{aligned} \text{queue_message} &= (45 * \text{num_msg_packets}) + 43 \mu\text{sec} \\ \text{retrieve_message} &= (61 * \text{num_msg_packets}) + 67 \mu\text{sec} \end{aligned}$$

These functions are closely related to a few of the overhead tasks described previously. The `queue_message` procedure is used by the application task to prepare a message for exchange over the AFTA optical network and to transfer the packets to the processor's local memory. The `send_queue` procedure is later activated by the rate group dispatcher and transfers the message packets from local memory to the dual port RAM buffers on the NE. It is important to maintain the distinction between these two operations, for `queue_message` occupies application task execution time while `send_queue` occupies RGD overhead time. The contributions of these two delays occur during different minor frames and therefore are not considered as a single delay entity, although the operations are actually performed on the same message packets. Similarly, `retrieve_message` is closely related to the scoop message operation included in the interrupt handler. The IH scoop transfers message packets from the NE message buffers to the processor's local memory; `retrieve_message` reconstructs the message from the packets stored in local memory and delivers it to the appropriate application task. Once again, these operations are performed on the same message packets but occur during different minor frames. Thus, the corresponding delays are considered independently.

8.3 Path Comparison Calculation

For each instantiation of an application task, the software analysis generates all possible execution paths through the appropriate task model. In order to determine which path represents the worst case delay, there must be a quantitative comparison of the individual paths, and the necessary calculations require the system specific delay data discussed in the previous section. As part of the path generation process, `generate_paths` calls upon `calculate_time` to produce a single delay value to represent each complete execution path. `calculate_time` uses the path parameterization produced by `parameterize` in combination with the list of constants derived from the performance measurement study to return a single integer value that represents the lower bound on execution delay for the given path. `generate_paths` then compares this value to the values for the previous execution paths to identify the worst case path.

`calculate_time` is essentially a simple series of algebraic manipulations that transforms a path parameterization into a tangible time value. At the beginning of the timing analysis, the `read_list` procedure develops a list of system specific constants

and coefficients by reading integer values from the file called “constants.dat” and storing them in the data structure called `delay_data`. All values are in terms of μsec and are derived from the AFTA benchmarking efforts. Note that these values are intentionally stored in an external file to be read at run-time so that any future changes to the benchmarking results do not force a recompilation of the timing analysis code. The integers held in `delay_data` correspond to the coefficients and constants listed in the delay equations of Section 8.2, and they also directly relate to the various elements of the path parameterization. Given the proper constants and the necessary path parameters, `calculate_time` simply applies the given equations to arrive at the lower bound on execution delay. The present list of path parameters is quite limited, and the same is true for the integer list held in `delay_data`; however, as the AFTA prototyping and benchmarking efforts progress, the parameter list will expand and the list of constants will grow accordingly. The timing analysis code is designed specifically to allow for such growth, and Appendix G outlines the process of adding new elements to the list of path parameters. As the AFTA project progresses and the timing analysis is revised, `calculate_time` will account for a broader range of delay factors and the values it returns will become more accurate.

8.4 Organizing Application Tasks

The fundamental goal of the hardware model analysis is to successfully integrate the results of the software analysis with the established system delay data in order to predict potential performance failures. Whereas the software analysis evaluates application tasks purely on an individual basis, the hardware analysis deals with tasks collectively according to their virtual group assignments. As such, the virtual group serves as the basic unit of analysis in this final phase, and the first objective is to properly categorize the application tasks according to the given system configuration.

The preliminary processing phase (Chapter 3) produces a file called “list_of_tasks.dat,” which contains the complete AFTA software configuration. The file is organized so that each task instantiation is listed on a single line along with the appropriate virtual group and rate group specifications and message passing limitations. Multiple instantiations of the same task are treated as distinct entities in order to simplify the task organization process and to allow for separate software analyses of the same task code with potentially different message passing limitations. The procedure called `process_list` is responsible for reading the configuration file and sorting the tasks into an array of 40 virtual groups. It is important to note that the software analysis is actually initiated within this sorting process, for as each task is taken from the

configuration file and placed in the proper VG, it is submitted to `task_parse`, which returns the worst case delay parameterization for the given task instantiation. Information about each virtual group is stored in a relatively extensive data structure; what follows is a detailed description of each element in the structure:

1. `present`: A boolean variable to indicate if a particular VG exists within the given system configuration. The AFTA allows up to 40 VGs within its configuration, but it is not likely that most configurations will utilize all the VGs. Therefore, this “presence bit” simplifies processing by signaling the analysis to ignore all non-existent VGs.
2. `overrun`: This is a one dimensional array of integers that is established by the `overrun_check` procedure. For every timing deadline that is not satisfied in the worst case analysis, a particular integer is stored in this array to indicate the nature of the performance failure. Section 8.4 explains this seemingly cryptic form of information storage.
3. `num_tasks`: An array of five integers used to establish the number of application tasks resident in each of the four rate groups on the given VG. Note that the fifth integer is simply a sum of the other four and thus indicates the total number of tasks associated with the virtual group.
4. `tasks`: An array of task-based data structures. For each task instantiation belonging to the virtual group, this structure stores the name, rate group specification, and worst case execution path parameterization. It is the development of this element of the virtual group structure that initiates the software analysis for each task in the suite.
5. `rg_totals`: An array of four integers that tracks the cumulative delay for tasks within each of the four rate groups. As a task is added to its respective virtual group, its worst case delay value is added to the total for its respective rate group. In other words, `rg_totals[1]` contains the cumulative worst case delay for all RG1 tasks within the VG.
6. `msg_totals`: A four element array of integer pairs that tracks the cumulative message traffic for each of the four rate groups. This information is required in calculating the overhead for each of the eight minor frames since the rate group dispatcher delay is a function of the rate group message traffic. As each task is added to the VG, its queued message parameters are added to the appropriate `msg_totals` integer pair. The first integer of the pair indicates the number of messages queued for the given rate group during a single execution cycle (under worst case conditions), and the second integer

denotes the total number of packets queued for that RG during one execution cycle. Note that message traffic is tracked only according to the number of messages queued, while the messages retrieved, sent, and read during an execution cycle are effectively ignored. Section 8.6 addresses the assumptions involved in focusing only upon queued message traffic.

Once the information in “list_of_tasks.dat” is exhausted, the virtual group records are considered complete, and the actual timing calculations begin.

8.5 Predicting Performance Failures

The procedure called `overrun_check` is responsible for the final timing calculations in the hardware analysis phase. The objective of these calculations is to predict potential failures to satisfy the hard-real-time constraints imposed by the AFTA’s rate group scheduling paradigm. In more concrete terms, the overrun check attempts to identify possible situations in which the combined delays of system overhead and application task execution exceed the time allotted within the second portion of any minor frame (refer to Figure 8-1).

Under the present rate group configuration, an 80ms time slice contains at least one full iteration of the tasks in all four rate groups, and as illustrated in Figure 8-2, it is comprised of 8 unique minor frames, each of which follows the execution pattern described in Section 8.2. Since the 80ms time slice contains at least one frame boundary for each rate group, all relevant timing constraints can be validated through a timing analysis that is limited to a set of 8 contiguous minor frames which spans a single RG1 frame, as shown below:

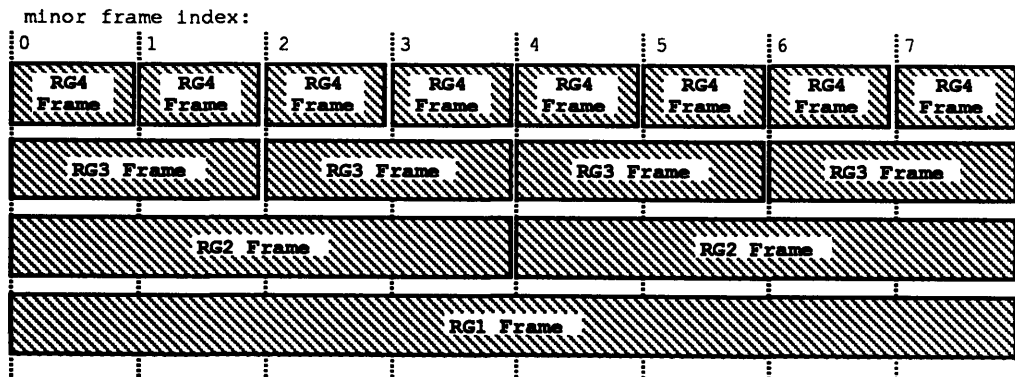


Figure 8-2, Rate Group Frame Organization

Since this analysis deals with worst case scenarios, it is naturally assumed that if timing deadlines are satisfied for a single complete time slice, they are satisfied for all

time. For every virtual group, the overrun check attempts to determine if, under worst case conditions, the tasks within each rate group can complete execution prior to the appropriate frame boundaries. Figure 8-3 (page 101) illustrates the deadlines that must be considered in such an analysis. For each minor frame, all overhead tasks and RG4 tasks within a given virtual group must complete a single execution cycle. Any remaining time within the minor frame is dedicated to scheduled tasks within the other rate groups. Notice that higher numbered rate groups have execution precedence over lower numbered groups, and until all tasks belonging to a higher priority rate group complete their execution cycles, tasks within lower priority rate groups are not granted any frame time. At each minor frame boundary, execution priority returns to the overhead and RG4 tasks. A mathematical summary of these timing constraints accompanies Figure 8-3 as a list of 15 inequalities. If all 15 inequalities are satisfied, the timing deadlines are satisfied.

Evaluating the inequalities themselves is a trivial task. What is important here is to explain the elements involved in the calculations and the rationale behind the inequalities. The first priority of `overrun_check` is to evaluate the delay due to system overhead in the second portion of each of the eight minor frames. As discussed in Section 8.2, this value is based on the sum of the following factors:

1. $IH_2 = (110 * \text{number_of_packets}) + 103 \text{ } (\mu\text{sec})$

2. $RGD_2 = \sum_{i=1}^{num_tsk} [(87 * num_pkt_i) + 27] + \sum_{i=1}^{num_rg_tsk} [(26 * num_rg_tsk_i) + 15] + 49 \text{ } (\mu\text{sec})$

3. $\text{context switches} = 19 \text{ } \mu\text{sec each}$

4. $\text{local FDIR} = 84 \text{ } \mu\text{sec}$

For each of the equations above, it is important to understand that this analysis assumes that rate group tasks always complete their execution cycles during the last minor frame prior to their respective frame boundaries. For instance, it is assumed that all RG1 tasks complete execution during minor frame 7, and it is assumed that all RG2 tasks complete execution cycles during minor frames 3 and 7. This type of assumption creates a worst case scenario for the system overhead because the burden of message passing operations imposed by the various rate groups is thereby concentrated in the minor frames that follow common frame boundaries, rather than being evenly distributed

among all eight minor frames. The same concept applies to the overhead imposed by context switching and the overrun checking performed by the rate group dispatcher.

For the first overhead equation shown above, the variable called *number_of_packets* is based upon the values stored in the `msg_totals` element of the VG record. In calculating the overhead for minor frame 0, *number_of_packets* is a sum of the packets queued by all four rate groups because it is assumed that all application tasks complete execution cycles during the previous minor frame. In a similar manner, when calculating the overhead for minor frame 6, *number_of_packets* is based upon the `msg_totals` values for RG3 and RG4 tasks, since both of these groups complete execution cycles during minor frame 5. The same principles apply in the second equation when calculating *num_pkt_i* and *num_tsk* for each minor frame. Also included in the rate group dispatcher equation is the overhead for overrun checking. This requires that within each minor frame, the RGD checks the status of all tasks that should have been completed within the previous minor frame. Figure 8-3 indicates how the frame boundaries affect various minor frames, and the information contained in the VG record is sufficient to determine the number of tasks to be checked by the RGD in a given minor frame. The evaluation of context switching overhead follows the same rules as used with the overhead for overrun checking. Lastly, local FDIR affects all minor frames and is appropriately added as a simple delay constant. As the minor frame overhead values are calculated, they are stored in an eight element array of integers called OH[.]

The first eight inequalities ensure that the RG4 tasks meet their timing deadlines in each of the eight minor frames. They simply compare the sum of overhead delay and RG4 task delay to the amount of time allotted within the second portion of the minor frame. The next group of four inequalities ensures that RG3 tasks complete their execution cycles without exceeding their frame boundaries. These calculations account for delays due to overhead and RG4 task execution to determine if there is sufficient frame time left over for the lower priority RG3 tasks. The next two inequalities are dedicated to validating the two frame boundaries encountered by RG2 tasks, and the last inequality ensures that RG1 tasks meet their timing deadlines. Notice that the calculations for each rate group account for delays due to all tasks with higher priorities while ignoring delays incurred by tasks with lower priorities. The inequalities are numbered 1 through 15, and the integer corresponding to each inequality that is not satisfied is stored in the `overrun` array of the VG record.

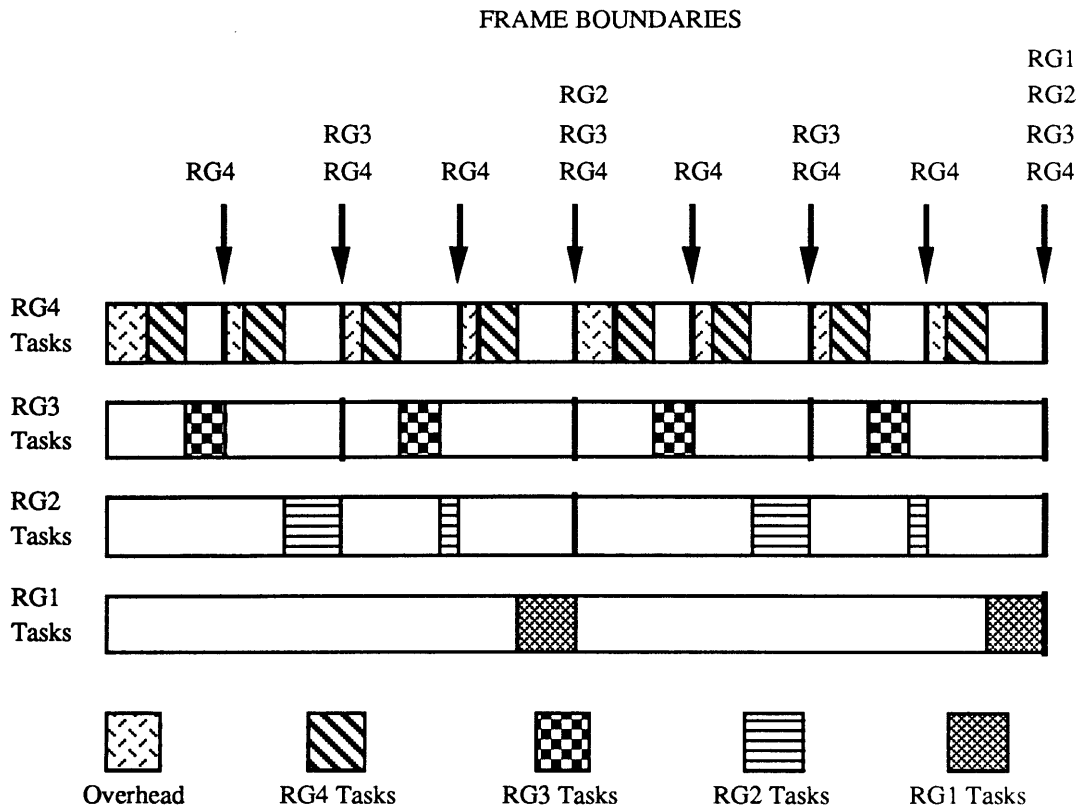


Figure 8-3, A Simplified View of Rate Group Scheduling

- 1: $OH(0) + RG4 < 1$ minor frame
- 2: $OH(1) + RG4 < 1$ minor frame
- 3: $OH(2) + RG4 < 1$ minor frame
- 4: $OH(3) + RG4 < 1$ minor frame
- 5: $OH(4) + RG4 < 1$ minor frame
- 6: $OH(5) + RG4 < 1$ minor frame
- 7: $OH(6) + RG4 < 1$ minor frame
- 8: $OH(7) + RG4 < 1$ minor frame
- 9: $OH(0) + OH(1) + 2xRG4 + RG2 < 2$ minor frames
- 10: $OH(2) + OH(3) + 2xRG4 + RG2 < 2$ minor frames
- 11: $OH(4) + OH(5) + 2xRG4 + RG2 < 2$ minor frames
- 12: $OH(6) + OH(7) + 2xRG4 + RG2 < 2$ minor frames
- 13: $OH(0) + OH(1) + OH(2) + OH(3) + 4xRG4 + 2xRG3 + RG2 < 4$ minor frames
- 14: $OH(4) + OH(5) + OH(6) + OH(7) + 4xRG4 + 2xRG3 + RG2 < 4$ minor frames
- 15: $OH(0) + OH(1) + OH(2) + OH(3) + OH(4) + OH(5) + OH(6) + OH(7) + 8xRG4 + 4xRG3 + 2xRG2 + RG1 < 8$ minor frames

$OH(x)$ refers to the overhead delay in minor frame x

RGx refers to the cumulative delay of all tasks belonging to rate group x

8.6 Assumptions

Throughout the AFTA timing analysis, a number of simplifying assumptions are introduced in an effort to produce useful results. It is important for the user to understand the nature of all assumptions made, for the accuracy of the analysis is significantly affected by the validity of the underlying assumptions. The following paragraphs discuss the assumptions that play a critical role in the integration of the application task models with the system specific delay data during the hardware analysis phase.

When the software analysis generates all possible execution paths through a particular code model, `calculate_time` is used to gauge the minimal worst case delay incurred by each path so that the paths can be effectively quantified and compared. In order to definitively state that Path A incurs a greater delay than Path B, the delay estimates for both paths must be based upon the same set of known deterministic quantities. This analysis assumes that for each critical construct included in the code model, an exact delay value can be added to the time total for an execution path which includes that construct. In other words, it is assumed that a task such as message retrieval incurs the same amount of delay every time it is called, regardless of the nature of the execution path. In the future, the AFTA will feature multiple architecture types for PEs, and in such a case, the delay for a message retrieval task would vary as a function of the VG on which a task resides. However, for any given task instantiation, the same critical construct always contributes the same delay for every possible execution path. The underlying assumption here is that problems such as network contention and bus contention have a negligible effect on the total delay for a critical construct. The results from the performance measurement study indicate relatively insignificant standard deviations for the operating system functions and message passing functions evaluated thus far. Problems may arise in the future as the AFTA's performance limits are tested and the global message traffic increases, but at this point it is safe to assume a high level of determinism in regard to those constructs presently included in the analysis. Also, VG phasing can be selected to minimize these effects.

A second assumption with regard to system delays is that the total time required for an execution path increases monotonically as a function of the frequency of occurrence for the critical constructs considered. In other words, an increase in the frequency of a critical construct within an execution path never results in a decrease in the total time delay incurred by that path; likewise, a decrease in the frequency of a construct does not result in an increase in time delay. This may seem to be an inherent assumption that need not be stated, but the coupling of overhead delays with application task delays through the message passing functions creates the possibility for unusual delay behavior

that should at least be addressed. It is possible that a particular combination of critical constructs within a worst case path could unexpectedly improve the efficiency of the overhead functions for the virtual group and thus transform a worst case path into a less than worst case path. Such unusual behavior would likely arise from some unforeseen system limitation which would prevent the overhead tasks from processing all given data and thus require less frame time. Fortunately, this type of non deterministic behavior has yet to appear in system testing, and the analysis presented here is conservative with respect to increases in efficiency.

The final assumption to be discussed in this section involves the nature of the global message traffic. For a static analysis, it is impossible to anticipate the number or the size of messages that are received by an application task or group of tasks within a given execution cycle. As such, it is assumed for this analysis that a VG only communicates with itself -- in other words, no inter-VG communication is considered. The result of this assumption is that for worst case evaluations, the number of message packets scooped for an execution cycle is set equal to the number of packets queued during that cycle. This type of assumption primarily affects overhead calculations, and its validity is critical since the per-packet processing time is quite significant in the delay calculation for the interrupt handler. The AFTA prototype testing has not yet addressed the detailed nature of global message traffic under various configurations, and thus the accuracy of this assumption cannot yet be gauged. Future system testing could call for some adjustments to the message scoop delay calculations.

8.6 Final Output File

The final output file for the AFTA timing analysis is called "results.dat." It is designed to present the results of both the software and hardware analysis an efficient and readable format. The results are organized according to virtual groups, and for each VG the following information is included:

1. An individual listing of each task instantiation that includes the task name, filename, rate group designation, and full worst case path parameterization.
2. For each of the four rate groups, a total worst case application task delay is listed to help the user recognize which rate groups are overburdened or underloaded.
3. For each of the eight minor frames, the total worst case overhead delay is listed.

4. A matrix of delay values lists the delay contribution from the overhead and each of the rate groups within each minor frame. This is intended to help the user identify potential timing problems and the sources of those problems.
5. A listing of all timing deadline violations as discovered through the evaluation of the 15 inequalities discussed earlier in this chapter.

The contents of the results file should reinforce the concept that this analysis tool is not valuable solely for the prediction of possible performance failures; rather, the overrun check is simply the most comprehensive result produced. The more important results are the intermediate values used in performing the overrun check. These include the overhead delay totals and the worst case path parameterizations of the individual application tasks. One of the major goals of this analysis is to properly characterize the software tasks for timing estimation, code optimization, and for further analysis of global message traffic and virtual group phasing. After a single configuration analysis, it should be readily apparent what types of changes need to be implemented to improve performance results. These changes might include streamlining application task code, changing the mapping between tasks and virtual groups, varying the VG configuration itself, or altering the rate group specification of one or more application tasks. As stated previously, the user always must consider the contents of the error file when evaluating the data in “results.dat,” for both sources of output from this analysis are relevant and should not be examined independently.

Chapter 9

Conclusions/Recommendations

9.1 Conclusions

This thesis presents an automated timing analysis tool that is designed specifically to characterize and evaluate the timing behavior of a given system configuration for the current AFTA prototype. The preceding chapters discuss a modular approach to the development of this automated tool, and the accuracy of the timing analysis depends upon the successful execution of each stage as well as the proper interaction between these stages. It is appropriate now to consider independently the effectiveness of each part of the analysis and then critique the usefulness of the analysis tool as a whole.

1. Preliminary processing is the first stage of the timing analysis. Its goal is to accurately describe the system's hardware and software configuration through an automated evaluation of the task specification file. This portion of the analysis is both simple and successful. No further development is required.
2. The accuracy of the analysis depends upon proper characterization of the system overhead in terms of both constant and variable delay elements. The AFTA performance measurement study addresses this task and presents the results to date, and the timing analysis tool depends upon these results for delay calculations and for the proper approach to those calculations. Given the state of the AFTA prototype, the overhead data is both accurate and thorough, but it will be necessary in the future to study the effects of I/O operations and network loading on system overhead. Such results must then be incorporated into the timing analysis that is presented here.
3. The accuracy of this analysis also depends upon the definition and measurement of significant sources of deterministic delays within application tasks. At this point, the estimated application task delays depend primarily upon the presence of message passing calls within the source code, and this focus is far too limited. In the future, the list of known deterministic delays or "critical constructs" must be expanded in order to better characterize the worst case timing behavior of individual application tasks. What is important here is that this tool currently includes all the necessary infrastructure for expanding the list of delay elements considered in a static analysis. At this stage of development, minimal marginal effort is required to expand the list of known deterministic delays, and this facilitates rapid improvement in the accuracy of the worst case delay estimates.

4. The second stage of the timing analysis consists of a static source code analysis for each application task in the suite, and the first phase of this analysis involves the development of source code models. At present, the modeling process is both tedious and delicate, but it stands as proof that an automated tool can effectively understand and evaluate the flow of execution through complex high level source code. The approach to code modeling is sound and allows for a great deal of flexibility in terms of the types of delay elements upon which to focus during a static analysis and the range of code constructs that can be analyzed successfully. Certainly, the modeling approach presented here is valid for other high level languages and could also be adapted to work with assembly language code. The primary weakness in this portion of the analysis tool is the lack of AFTA application task source code with which to test the modeling process. The intricacies of the code parsing and modeling tools create a great deal of room for minor errors, and it is important to subject these tools to rigorous testing with any relevant source code that becomes available.
5. The second phase of the software analysis involves the quantitative evaluation of the source code models constructed for the various application tasks. Since the model conventions are well-defined and limited by design, this portion of the timing analysis lends itself to extensive testing and improvement. Given any valid model, the model analysis phase successfully finds all possible execution paths and then parameterizes, quantifies, and compares them in order to identify the worst case path. This part of the analysis tool is by far the most successful, and the principles involved and the algorithms developed can easily be adapted to future changes in the modeling conventions or the path quantification process.
6. The final stage of the analysis involves the integration of the individual task analyses in a manner that reflects the system virtual group configuration and accurately characterizes the amount of variable minor frame overhead incurred by the given task groupings. This part of the analysis is relatively straight forward and quite robust, but it is not complete. Due to the absence of I/O overhead data, the calculations of the hardware model analysis are not completely accurate. It should be a simple task to later add the I/O measurement results to the hardware model used by this analysis tool, but until then, one must remember that the results produced are somewhat incomplete.

As a whole, the analysis tool presently produces results of limited utility, but both the code infrastructure and the approach to an automated timing analysis are robust and very valuable. All the necessary elements of a successful analysis tool are present, but there is a need for further development and testing.

9.2 Recommendations for Further Study

The following list details recommendations for further development of the automated timing analysis tool:

1. In order to improve the accuracy to the worst case estimates, the list of critical constructs must be expanded. As more application task source code becomes available, it should be easy to determine what types of functions and Ada constructs significantly contribute to the execution delay of application tasks. Another good source of critical constructs would be the standard Ada libraries. If one were to benchmark all the functions included in those libraries, any standard Ada function could be accounted for in the software analysis.
2. As application task source code is developed for the AFTA, it should be subjected to the code modeling tools presented here. Extensive testing will expose some minor errors in the code modeling process, and it is important to correct these problems as soon as possible.
3. When the I/O portion of the AFTA operating system is fully developed and tested, its timing characteristics should be measured, modeled, and incorporated into the hardware model analysis. The accuracy of the system overhead calculations depends upon knowledge of I/O operations, and any enhancements to these calculations will improve the usefulness of the analysis results. Likewise, any I/O operations that typically appear in the application task source code should be quantified and added to the list of critical constructs.
4. Another useful improvement to the overhead calculations involves proper characterization of the system's global message passing operations. Presently it is assumed that a task receives the same amount of message traffic that it sends, and it would be useful to either validate this assumption or develop a more accurate model of global message traffic to be incorporated into the overhead calculations.

Appendix A HEADER.H

```
/* This is a common header that is utilized by both START.C and FINISH.C */

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

/* The following constants identify critical constructs and also serve as */
/* indices to the search_list array. The code refers to critical constructs */
/* and search_list entries according to the names shown below. */
#define WFS 0
#define LOOP 1
#define IF 2
#define ELSE 3
#define ELSIF 4
#define CASE 5
#define WHEN 6
#define QUEUE 7
#define RETRIEVE 8
#define SEND 9
#define READ 10
#define TASK 11
#define EQUAL 12
#define RANGE 13
#define PACKAGE 14
#define PROC 15
#define BEGIN 16
#define END 17
#define GTID 18
#define UG 19
#define RG 20
#define XMITSIZE 21
#define XMITNUM 22
#define RCUESIZE 23
#define RCUENUM 24
#define SELECT 25
#define RECORD 26
#define ACCEPT 27

/* These constants are for model entries only. There are no */
/* corresponding entries in search_list */
#define END_LOOP 51
#define END_IF 52
#define END_CASE 53
#define COUNTERSET 54
#define FOR_LOOP 55
#define WHILE_LOOP 56

/* Defines the various pertinent classes to which a string may belong */
#define UNKNOWN 0
#define SIMPLENUM 1
#define NATURALNUM 2
#define COMPLEXNUM 3
#define RANGENUM 4
```

Appendix A

```
#define UARNAME 5

/* These constants describe the status of the processing */
#define BLANK 0
#define LOST -1
#define UNDEFINED -2
#define INFINITE -3
#define DEFAULT -4
#define EXEC -5
#define NO_EXEC -6

/* Define the minor frame time here */
#define MINORFRAME 10000

enum boolean {NO = 0, YES = 1};

struct msg_pair
{
    int num_msg_queued;
    int num_packets;
};

struct var_info
{
    char name[100];
    int value;
};

struct var_list_info
{
    struct var_info entry[100];
    int length;
    int marker;
};

struct string
{
    char name[100];
};

struct list_info
{
    struct string entry[100];
    int marker;
    int length;
};

struct match_info
{
    char name[100];
    char filename[100];
};

struct counter_list
{
    int num_sent;
    int num_read;
    int num_queued;
    int num_retrieved;
};
```

Appendix A

```
    int num_msg_queued;
    int msg_retrieved;
    int total_time;
};

struct task_parse_info
{
    char name[100];
    int rate_group;
    struct counter_list counter_set;
};

struct message_limits
{
    int xmit_size;
    int xmit_num;
    int rcve_size;
    int rcve_num;
};

struct task_spec_info
{
    char name[100];
    int virtual_group;
    int rate_group;
    struct message_limits limits;
};

struct ug_info
{
    enum boolean present;
    int overrun[20];
    int num_tasks;
    struct task_parse_info task[20];
    int rg_total[5];
    int overhead[8];
    struct msg_pair msg_total[5];
};

struct range_info
{
    char description[100];
    int first;
    int last;
    int span;
};

struct flag_list
{
    enum boolean task_found;
    enum boolean pkg_found;
    enum boolean ctr_active;
    enum boolean finished;
    enum boolean fatal_error;
    enum boolean enable_when;
    int proc_depth;
};

struct comment_info
```


Appendix A

```
{
    int basic_loop_limit;
    int for_loop_limit;
    int while_loop_limit;
    int message_size;
};

struct model_entry_info
{
    int type;
    int value;
    int depth;
    int pointer;
    int flow;
};

struct model_info
{
    int length;
    int num_counters;
    struct model_entry_info entry[100];
    struct counter_list counter_set[100];
};

struct proc_info
{
    char name[100];
    char filename[100];
    enum boolean done;
    struct model_info skeleton;
};

struct proc_list_info
{
    struct proc_info entry[100];
    int length;
    int marker;
    int pkg_marker;
};

struct constant_list
{
    int queue_coeff;
    int queue_const;
    int retrieve_coeff;
    int retrieve_const;
    int IH_coeff;
    int IH_const;
    int RGD_msg_coeff;
    int RGD_msg_const;
    int RGD_tsk_coeff;
    int RGD_tsk_const;
    int RGD_overall_const;
    int RGD_empty_queue_const;
    int context_switch;
    int local_FDIR;
    int sys_FDIR;
};
```

Appendix B START.C

```
#include "header.h"

/*-----
CALLNAME: START.C
AUTHOR: S. Treadwell
CREATED: 19 MAY 92
UPDATED: 20 JUL 92

This is the preliminary processing code. It opens up the task specification
file and processes the contents. All required system configuration info is
stored in two output files "task_names.dat" and "list_of_tasks.dat" and
passed on to future stages in the analysis.
-----*/

main()
{
    struct task_spec_info task[20]; /* temporarily stores all config info */
    struct list_info search_list;
    struct list_info this_line;
    FILE *infile;
    int i,c;
    int signal = 1;
    int num_tasks = 0;

    /* Establish the list of key words that control the search for config info */
    read_list(&search_list);

    /* Open the task specification file */
    infile = fopen("task_list.ada","r");
    if (infile == NULL)
    {
        printf("Cannot open task_list_io.ada \n");
        exit (2);
    }

    /* Search the file line by line and extract relevant info */
    do
    {
        signal = get_line(&this_line,infile);
        search(&search_list,&this_line,task,&num_tasks);
    }
    while (signal != 0);

    fclose(infile);

    /* Produce the output files to be used by the DCL search and the remainder */
    /* of the analysis */
    write_file(task,num_tasks);
}

```

Appendix B

```
/*-----*/
CALLNAME: READ_LIST
AUTHOR: S. Treadwell
CREATED: 13 APR 92
UPDATED: 15 JUN 92

This procedure establishes the list of key words that control the search
for configuration information. The list is taken from "key_words.dat"
and is stored in search_list. Each key word occupies a separate line in
the external file.
-----*/

read_list(search_list)
struct list_info *search_list;
{
    FILE *list_file;
    int signal = 0;
    int value = 0;
    int num = 0;

    list_file = fopen("key_words.dat","r");
    if (list_file == NULL)
    {
        printf("Cant open key_words.dat for input\n");
        exit(2);
    }

    while(signal != EOF)
        signal = fscanf(list_file,"%s\n",search_list->entry[num++].name);

    search_list->length = num - 1;
    fclose(list_file);
}
```

Appendix B

```
/*-----  
CALLNAME: GET_LINE  
AUTHOR: S. Treadwell  
CREATED: 22 APR 92  
UPDATED: 10 JUN 92  
  
Will use the stream of characters provided by the specified file to assemble  
a buffer of single word items all of which belong to a single line of code.  
The key here is to gather words until a carriage return is found.  
Blank lines are not recorded in the buffer. Comment lines are recorded as  
a single word denoted as '--'; all additional words on a comment line are  
ignored. This procedure will not record a line of words if it is terminated  
by a EOF character.  
  
-----*/  
  
int get_line(this_line,infile)  
struct list_info *this_line;  
FILE *infile;  
{  
    int c,k;  
    int num = 0;  
    int count = 0;  
    int comment = NO;  
    int signal = 1;  
    char temp[80];  
  
    while(isspace(c = fgetc(infile))); /* bleed off white space */  
    do  
    {  
        if (c == EOF) break;  
        do  
            temp[count++] = c; /* append character to word */  
        while(!isspace(c = fgetc(infile)));  
        temp[count] = '\0';  
        strcpy(this_line->entry[num].name,temp); /* append word to line */  
  
        do  
            if (c == '\n') break;  
        while(isspace(c = fgetc(infile)));  
  
        count = 0;  
        if ((k = strcmp(temp,"--")) == 0) comment = YES;  
        if (comment == NO) ++num;  
    }  
    while (c != '\n');  
  
    if (c == EOF) signal = 0;  
    this_line->length = num;  
    this_line->marker = 0;  
  
    return(signal);  
}
```

Appendix B

```
/*-----
CALLNAME: SEARCH
AUTHOR: S. Treadwell
CREATED: 11 APR 92
UPDATED: 22 JUN 92

Will do the primary parsing for the "start" stage. Once a line of code is
available for analysis, 'search' will inspect it for a given list of ada
constructs and system calls and will branch to other parsing and analysis
functions as dictated by what is found.
This procedure looks for call names and ada constructs exactly as given by
the user in auxillary files. When searching for matches, names that are
similar but not exactly the same as the names sought will not be
sufficient to warrant a match.
-----*/

search(search_list,this_line,task,num_tasks)
struct list_info *search_list;
struct list_info *this_line;
struct task_spec_info task[];
int *num_tasks;
{
    int i,j,k;
    int size;

    for(j = 0; j < search_list->length; ++j)
        for(i = 0; i < this_line->length; ++i)
            if((k= strcmp(this_line->entry[i].name,search_list->entry[j].name)) == 0)
                {
                    this_line->marker = i + 2;
                    switch(j)
                    {
                        case GTID: /* extract the task name */
                            extract_name(this_line,task,num_tasks);
                            break;
                        case UG: /* extract the task's virtual group assignment */
                            task[*num_tasks].virtual_group = strip(this_line->entry[i+2].name);
                            break;
                        case RG: /* extract the task's rate group designation */
                            task[*num_tasks].rate_group = get_rg(this_line->entry[i+2].name);
                            break;
                        case XMITSIZE: /* extract message passing limitations */
                            task[*num_tasks].limits.xmit_size = strip(this_line->entry[i+2].name);
                            break;
                        case XMITNUM:
                            task[*num_tasks].limits.xmit_num = strip(this_line->entry[i+2].name);
                            break;
                        case RCUESIZE:
                            task[*num_tasks].limits.rcue_size = strip(this_line->entry[i+2].name);
                            break;
                        case RCVENUM:
                            task[*num_tasks].limits.rcue_num = strip(this_line->entry[i+2].name);
                            *num_tasks += 1;
                            break;
                        default: break;
                    }
                }
    }
}
```

Appendix B

```
/*-----  
CALLNAME: WRITE_FILE  
AUTHOR: S. Treadwell  
CREATED: 26 MAY 92  
UPDATED: 12 JUN 92  
  
This procedure is called at the end of the "start" stage. It will produce  
two output files for use by later stages. The first file contains names of  
the tasks implemented in the task suite; it is used by the DCL search in  
finding the files where the tasks reside. The second file contains all  
pertinent info about the task suite, and it is used by the "finish" stage in  
parsing the tasks and calculating overruns.  
  
-----*/  
  
write_file(task,num_tasks)  
struct task_spec_info task[];  
int num_tasks;  
{  
    int i,j,k;  
    int repeat;  
    FILE *outfile;  
  
    outfile = fopen("list_of_tasks.dat","w");  
    if (outfile == NULL)  
    {  
        printf("Cannot open list_of_tasks.dat\n");  
        exit(2);  
    }  
  
    for(i = 0; i < num_tasks; ++i)  
        fprintf(outfile,"%s %d %d %d %d %d %d\n",  
            task[i].name,task[i].virtual_group,task[i].rate_group,  
            task[i].limits.xmit_size,task[i].limits.xmit_num,  
            task[i].limits.rcve_size,task[i].limits.rcve_num);  
    close(outfile);  
  
    outfile = fopen("task_names.dat","w");  
    if(outfile == NULL)  
    {  
        printf("Cannot open the output file\n");  
        exit(2);  
    }  
  
    for(i = 0; i < num_tasks; ++i)  
    {  
        repeat = NO;  
        for(j = 0; j < i; ++j)  
            if ((k = strcmp(task[i].name,task[j].name)) == 0)  
                repeat = YES;  
        if (repeat == NO)  
            fprintf(outfile,"%s\n",task[i].name);  
    }  
  
    fprintf(outfile,"done\n");  
  
    fclose(outfile);  
}
```

Appendix B

```
/*-----  
CALLNAME: EXTRACT_NAME  
AUTHOR: S. Treadwell  
CREATED: 26 MAY 92  
UPDATED: 09 JUN 92  
  
Used to extract task name from the task specification file.  
-----*/  
  
extract_name(this_line,task,num_tasks)  
struct list_info *this_line;  
struct task_spec_info task[];  
int *num_tasks;  
{  
    char temp[40];  
    char task_name[40];  
    int count = 6;  
    int count2 = 0;  
    int num;  
  
    num = this_line->marker;  
    strcpy(temp,this_line->entry[num].name);  
  
    /* eliminate the prefix "gtids." from the task name */  
    while (temp[count] != ',')  
        task_name[count2++] = temp[count++];  
    task_name[count2] = '\0';  
  
    /* Adhere to the naming convention for task names */  
    strcat(task_name,"_t");  
    strcpy(task[*num_tasks].name,task_name);  
}
```

Appendix B

```
/*-----  
CALLNAME: STRIP  
AUTHOR: S. Treadwell  
CREATED: 26 MAY 92  
UPDATED: 12 JUN 92  
  
Used to take the comma off the end of a number and convert it from a string  
to an integer.  
-----*/
```

```
int strip(argument)  
char argument[];  
{  
    int count = 0;  
    int value = 0;  
    char temp[20];  
  
    /* strip the comma off the end */  
    while(argument[count] != ',')  
    {  
        temp[count] = argument[count];  
        count++;  
    }  
    temp[count] = '\0';  
  
    /* convert the string to an integer value */  
    value = strtol(temp,(char **)NULL,10);  
    return(value);  
}
```

```
/*-----  
CALLNAME: GET_RG  
AUTHOR: S. Treadwell  
CREATED: 26 MAY 92  
UPDATED: 12 JUN 92  
  
This extracts the rate group number for a task from the task spec file.  
-----*/
```

```
int get_rg(argument)  
char argument[];  
{  
    char rg[10];  
  
    /* grab the rg * directly and convert it to an integer */  
    rg[0] = argument[9];  
    rg[1] = '\0';  
  
    return(strtol(rg,(char **)NULL,10));  
}
```


Appendix C

DCL Source Code

ANALYZE.COM

This program orchestrates the entire analysis process from start to finish.

```
$run start
$@find
$run finish
$exit
```

FIND.COM

This program finds the files that hold the application task source code and produces "filenames.dat" as output.

```
$task = "task"
$body = "body"
$is = "is"
$!search /window=0 /output=temp.dat /match=and-
$! [ftpp.afta.source.cmslib...].ada-
$! 'task','body','is'
$search /window=0 /output=temp.dat /match=and-
 [treadwell.afta...].ada-
 'task','body','is'
$open/append outfile temp.dat
$write outfile "done"
$close outfile
$type temp.dat
$open/write outfile filenames.dat
$open/read taskfile task_names.dat
$bigloop:
$read taskfile taskname
$if taskname .eqs. "done" then goto finished
$write outfile taskname
$open/read infile temp.dat
$loop:
$read infile filename
$if filename .eqs. "done" then goto done
$search 'filename' 'taskname'
$match = $severity .eq. 1
$if match then write outfile filename
$goto loop
$done:
$close infile
$goto bigloop
$finished:
$close outfile
$close taskfile
$type filenames.dat
$!purge
$exit
```


Appendix D

FINISH.C

```
*include "header.h"

/* This is the controlling "main" for finish.exe */

main()
{
    void match_up();
    void process_list();
    void check_overrun();
    void write_file();
    struct match_info task[40];
    struct vg_info vg[40];
    struct constant_list delay_data;
    int i,n;
    int num_tasks;
    FILE *error_file;

    error_file = fopen("errors.dat","w");
    if(error_file == NULL)
    {
        printf("Cannot open errors.dat for output\n");
        exit(2);
    }

    match_up(task,&num_tasks,error_file);

    /* The following info tells the user which version of the task suite is */
    /* being analyzed. Note it's recorded in the error log rather than the */
    /* results file. */
    fprintf(error_file,"The matching between tasks and filenames is...\n");
    for (i = 0; i < num_tasks; ++i)
        fprintf(error_file,"%d %s %s\n",i,task[i].name,task[i].filename);

    process_list(vg,task,num_tasks,&delay_data,error_file);

    check_overrun(vg,&delay_data,error_file);

    write_file(vg);

    fclose(error_file);
}
```

Appendix D

```

/*-----
CALLNAME: MATCH_UP
AUTHOR: S. Treadwell
CREATED: 27 MAY 92
UPDATED: 25 JUN 92

Takes the output file from the DCL task body search and performs a matchup
between task names and the files in which they reside.  If the task was not
found, it is matched with a "none."  If the task name was found in multiple
files, the user is given notice and the choice of which file he wants to
have processed for that task.  This gives the user the choice of which
version of a task he wants to have analyzed since it is assumed that several
versions of the same task may be present in the target directory.
-----*/

void match_up(task,num_tasks,error_file)
struct match_info task[];
int *num_tasks;
FILE *error_file;
{
    struct string file[10];
    int i = 0;
    int j = 0;
    int num = 0;
    int choice = 0;
    int signal = 0;
    int k,n,c;
    char dummy[80];
    FILE *infile;

    infile = fopen("filenames.dat","r"); /* File produced by the DCL search */
    if (infile == NULL)
    {
        fprintf(error_file,"Cannot open filenames.dat for input\n");
        exit(2);
    }

    /* Data format: task name followed by corresponding filename(s) with each */
    /* name listed on a separate line.  The first task name is grabbed and */
    /* converted to typical string format. */
    fscanf(infile,"%s\n",task[i].name); strcat(task[i++].name,"\0");

    /* Now, corresponding filename(s) and additional task names are grabbed */
    /* from the file and converted to string format.  Whenever a new task name */
    /* is found, the filename(s) for the previous task get(s) processed.  Note */
    /* that 0,1, or multiple filenames (up to 10) can be handled for each task */
    do
    {
        signal = fscanf(infile,"%s\n",dummy);  strcat(dummy,"\0"); /* grab name */
        num = strlen(dummy);
        if((((c = dummy[num-1]) == 't') && ((c = dummy[num-2]) == '_')) ||
            (signal == EOF)) /* Is it a task name? or the end of the file? */
        {
            switch(j) /* Process filename(s) for last task name */
            {
                case 0: strcpy(task[i-1].filename,"none"); break;
                case 1: strcpy(task[i-1].filename,file[0].name); break;
                default: /* Let the user choose which version of the task to analyze */

```

Appendix D

```
printf("Multiple files found for the task called %s\n",task[i].name);
for (n = 0; n < j; ++n)
    printf("%d : %s \n",n,file[n].name);
printf("Your Choice: "); scanf("%d",&choice);
if(choice < j)
    strcpy(task[i-1].filename,file[choice].name);
}
if (signal != EOF)
{
    strcpy(task[i].name,dummy); /* If it was a task name, store it */
    ++i; j=0;
}
}
else
    strcpy(file[j++].name,dummy); /* If not, store it as a filename */
}
while(signal != EOF);

*num_tasks = i;

/* Procedure ends with an array of paired names stored in task[]. Each */
/* task name has a single filename associated with it (or "none") and */
/* num_tasks reveals how many tasks are included in the analysis */
}
```

Appendix D

```
/*-----
CALLNAME: PROCESS_LIST
AUTHOR: S. Treadwell
CREATED: 09 JUN 92
UPDATED: 14 OCT 92

Takes the output file from start.exe and gleans from it all the necessary
information about the task suite that is contained in the task specification
file. The information is then grouped and stored according to the vgs in
which the tasks reside. Note that task parsing and analysis is initiated
here, and the results stored in vg[] are later submitted to check_ouerrun
for an integrated hardware and software configuration analysis.
-----*/

void process_list(vg,task,num_tasks,delay_data,error_file)
struct vg_info vg[];
struct match_info task[];
int num_tasks;
struct constant_list *delay_data;
FILE *error_file;
{
    void read_list();
    struct counter_list task_parse();
    struct message_limits messages;
    struct list_info search_list;
    int i,j,k;
    int num;
    int num_loaded;
    int marker;
    int vgnum;
    int rgnum;
    int signal;
    char dummy[80];
    char task_name[80];
    FILE *infile;

    infile = fopen("list_of_tasks.dat","r"); /* File produced by start.exe */
    if(infile == NULL)
    {
        fprintf(error_file,"Cannot open list_of_tasks.dat for input\n");
        exit(2);
    }

    /* Establish the array of key words that will be used in the software */
    /* modeling and analysis phase. Also process the system specific delay */
    /* values and store them in delay_data. */
    search_list.length = 0;
    read_list(&search_list,delay_data);

    while(1)
    {
        /* Grab information relating to a single instantiation of a single task */
        /* Each task instantiation is given a single line in the input file */
        /* The format should be readily apparent from the fscanff statement below */
        signal = fscanf(infile,"%s %d %d %d %d %d %d\n",task_name,&vgnum,&rgnum,
            &messages.xmit_size,&messages.xmit_num,
            &messages.rcve_size,&messages.rcve_num);
    }
}
```

Appendix D

```
if (signal == EOF) break; /* End of file terminates the procedure */

/* Find the filename for the task that is to be analyzed and mark it */
for(j = 0; j < num_tasks; ++j)
    if((k = strcmp(task_name,task[j].name)) == 0) marker = j;

/* Update appropriate vg record according to the task instantiation */
/* Note that each task instantiation is processed individually so that */
/* different message passing limitations can be analyzed for a single */
/* task. */
num = vg[vgnum].num_tasks;
vg[vgnum].present = YES; /* Yes, this vg* exists in this configuration */
vg[vgnum].num_tasks += 1;
strcpy(vg[vgnum].task[num].name,task_name);
vg[vgnum].task[num].rate_group = rgnum;

/* Now kick off the analysis of the specified task with the call to */
/* task_parse. The analysis culminates in the establishment of a single */
/* counter_set that will represent the specified task instantiation */
/* during the integrated hardware and software analysis in check_ouerrun */
fprintf(error_file,"\fSOFTWARE ANALYSIS FOR %s\n\n",task[marker].name);
vg[vgnum].task[num].counter_set =
    task_parse(&search_list,delay_data,task[marker].name,
              task[marker].filename,messages,error_file);
}

fclose(infile);

/* This procedure completes with vg[] holding all config information as */
/* well as all worst case execution delay analysis results for each */
/* instantiation of every task in the suite. */
}
```


Appendix D

```
/*-----  
CALLNAME: TASK_PARSE  
AUTHOR: S. Treadwell  
CREATED: 10 AUG 92  
UPDATED: 14 OCT 92  
  
This procedure has responsibility for modeling and analyzing a single task  
to find a parameterized expression for the worst case execution path through  
the task for a single execution cycle. The parameterization is returned to  
the parent procedure.  
-----*/  
  
struct counter_list task_parse(search_list,delay_data,task_name,filename,  
                               messages,error_file)  
  
struct list_info *search_list;  
struct constant_list *delay_data;  
char task_name[];  
char filename[];  
struct message_limits messages;  
FILE *error_file;  
{  
    void find_packages();  
    void parse();  
    void print_procedures();  
    struct counter_list find_worst_path();  
    struct list_info pkg_list;  
    struct model_info skeleton;  
    struct counter_list final_counter;  
    struct proc_list_info procedures;  
    struct counter_list counter_set;  
    struct flag_list flags;  
    char pkg_filename[80];  
    char pkg_name[80];  
    int i,k;  
  
    /* Set a flag in the error log to help establish the chronology of the */  
    /* analysis and any errors. */  
    fprintf(error_file,"Working on %s\n",task_name);  
  
    /* Initialize the worst case parameterization for the task */  
    final_counter.num_queued = 0;  
    final_counter.num_retrieved = 0;  
    final_counter.num_sent = 0;  
    final_counter.num_read = 0;  
  
    if ((k = strcmp(filename,"none")) == 0)  
    {  
        fprintf(error_file,"No file was found for task %s\n",task_name);  
        goto doneparsing;  
    }  
  
    /* Prepare for bottom-up modeling */  
    pkg_list.length = 0;  
    procedures.length = 0; procedures.marker = 0; procedures.pkg_marker = 0;  
    skeleton.length = 0; skeleton.num_counters = 0;  
  
    /* Establish the task's procedure/package hierarchy */  
    find_packages(filename,&pkg_list,error_file);  
}
```

Appendix D

```
/* Starting at the bottom of the package list, each package is processed */
/* Individually to collect procedure models in preparation for modeling */
/* the actual task body. */
for (i = pkg_list.length - 1; i >= 0; --i)
{
    strcpy(pkg_filename,pkg_list.entry[i].name);
    strcat(pkg_filename, ".ada"); /* Filename is derived from package name */
    parse(search_list,delay_data,pkg_filename,pkg_list.entry[i].name,task_name,
          messages,&procedures,&skeleton,&flags,error_file);
}

/* Now parse model the task body since all supporting procedures */
/* have already been examined */
strcpy(pkg_name,"none");
parse(search_list,delay_data,filename,pkg_name,task_name,messages,
      &procedures,&skeleton,&flags,error_file);

/* Record info in the error log to help with any debugging needed */
print_procedures(&procedures,error_file);

/* Now analyze the resulting task model to find the worst case exec path */
/* assuming that the task body was found in the specified filename */
if(flags.task_found)
{
    final_counter = find_worst_path(&skeleton,delay_data,error_file);
    /* Record results in output file */
    fprintf(error_file,"The worst path is characterized by...\n");
    fprintf(error_file,"Queued: %3d Retrieved: %3d Sent: %3d Read: %3d \n",
            final_counter.num_queued,final_counter.num_retrieved,
            final_counter.num_sent,final_counter.num_read);
}
else
    fprintf(error_file,"The task %s was not found in %s\n",task_name,filename);

doneparsing: return(final_counter);

/* final_counter holds the worst case path parameterization and is stored */
/* by the parent procedure (process_list) in the appropriate ug record as */
/* the definitive representation of the given task instantiation */
}
```

Appendix D

```
/*-----  
CALLNAME: FIND_PACKAGES  
AUTHOR: S. Treadwell  
CREATED: 12 JUL 92  
UPDATED: 15 OCT 92  
  
This procedure builds the structural hierarchy for a given task by examining  
its context clause for package names upon which the task depends. It also  
examines context clauses for all supporting packages to find packages upon  
which they depend. The result is a one-dimensional array of package names  
with the most fundamental packages at the bottom of the list -- in other  
words, the packages toward the top of the list depend upon the ones at the  
bottom of the list.  
-----*/  
  
void find_packages(filename, pkg_list, error_file)  
char filename[];  
struct list_info *pkg_list;  
FILE *error_file;  
{  
    void update_pkg_list();  
    int i, j;  
    char pkg_filename[80];  
  
    /* Look at the context clause in the package containing the task body */  
    update_pkg_list(filename, pkg_list, error_file);  
  
    /* Look at the context clause in all packages contained in package list */  
    /* Any new packages found are appended to the list, and the search */  
    /* continues in a recursive manner until reaching the end of the package */  
    /* list. */  
    for(i = 0; i < pkg_list->length; ++i)  
    {  
        strcpy(pkg_filename, pkg_list->entry[i].name);  
        strcat(pkg_filename, ".ada");  
        update_pkg_list(pkg_filename, pkg_list, error_file);  
    }  
  
    /* Record results in error file purely for debugging purposes */  
    fprintf(error_file, "The packages found are...");  
    if(pkg_list->length == 0) fprintf(error_file, "none");  
    fprintf(error_file, "\n");  
    for(i = 0; i < pkg_list->length; ++i)  
        fprintf(error_file, "%s\n", pkg_list->entry[i].name);  
}
```

Appendix D

```
/*-----
CALLNAME: UPDATE_PKG_LIST
AUTHOR: S. Treadwell
CREATED: 15 JUL 92
UPDATED: 18 NOV 92

This procedure examines the context clause of a given package in order to
find names of packages upon which the given package depends. This is a
method of exploring the subprogram hierarchy for a given task, and all new
package names found in the context clause are appended to the package list
to be later examined by this procedure.
-----*/

void update_pkg_list(filename, pkg_list, error_file)
char filename[];
struct list_info *pkg_list;
FILE *error_file;
{
    void with_found();
    struct list_info this_line;
    struct list_info comment_line;
    struct comment_info info_buffer;
    int i, j, k;
    int signal = 1; /* Serves to signal the end of the file */
    FILE *infile;

    infile = fopen(filename, "r");

    if(infile != NULL)
        while (signal != 0)
        {
            /* Search the file looking for a program statement of the form ... */
            /* "with [PKG_NAME];" When such a statement is found, the package */
            /* name is extracted and checked against the package list to */
            /* determine if it should be appended to the current list */
            signal=get_line(&this_line, &comment_line, &info_buffer, infile, error_file);
            if((k = strcmp(this_line.entry[0].name, "with")) == 0)
            {
                this_line.marker = 1;
                with_found(&this_line, pkg_list, infile);
            }
        }

    fclose(infile);
}
}
```

Appendix D

```
/*-----  
CALLNAME: PARSE  
AUTHOR: S. Treadwell  
CREATED: 10 JUL 92  
UPDATED: 18 NOV 92  
  
A generic code processing procedure that accepts code from all types of  
program units -- packages, subprograms, and tasks. For subprograms, it  
constructs code models and adds the model info to the procedures data  
structure. When dealing with a task body, the task model is developed and  
returned to task_parse in the skeleton data structure.  
-----*/  
  
void parse(search_list,delay_data,filename,pkg_name,task_name,messages,  
          procedures,skeleton,flags,error_file)  
struct list_info *search_list;  
struct constant_list *delay_data;  
char filename[];  
char pkg_name[]; /* Passed as "none" when parsing the task body file */  
char task_name[];  
struct message_limits messages;  
struct proc_list_info *procedures; /* Holds all subprogram models */  
struct model_info *skeleton; /* Always holds the current model */  
struct flag_list *flags; /* Provides important status information */  
FILE *error_file;  
{  
    int get_line();  
    void search();  
    struct list_info this_line; /* Holds the current program statement */  
    struct list_info comment_line; /* Used to grab vital programmer input */  
    struct list_info end_list; /* Needed for status info */  
    struct comment_info info_buffer; /* Holds current summary of prog. input */  
    int signal = 1; /* Signals when the end of the given file is reached */  
    int i,k;  
    FILE *infile; /* Ada source file that is currently being parsed */  
    FILE *outfile; /* Error log */  
  
    /* Set-up */  
    end_list.length = 0;  
    comment_line.length = 0;  
    this_line.length = 0;  
  
    flags->ctr_active = NO;  
    flags->pkg_found = NO;  
    flags->task_found = NO;  
    flags->finished = NO;  
    flags->fatal_error = NO;  
    flags->enable_when = NO;  
    flags->proc_depth = 0;  
  
    info_buffer.basic_loop_limit = INFINITE;  
    info_buffer.for_loop_limit = UNDEFINED;  
    info_buffer.while_loop_limit = UNDEFINED;  
    info_buffer.message_size = DEFAULT;  
  
    /* Note that it is assumed that all packages and files upon which a task */  
    /* depends are included in the same directory as the task body file. */  
    infile = fopen(filename,"r");
```

Appendix D

```
if(infile == NULL)
{
    fprintf(error_file,"File %s could not be found\n",filename);
    goto done_parsing;
}

/* Help specify the chronology of the error log by listing the file that */
/* is currently being processed */
if((k = strcmp(pkg_name,"none")) == 0) /* Is this the actual task file? */
    fprintf(error_file,"Now processing task %s\n",task_name);
else
    fprintf(error_file,"Now processing package %s\n",pkg_name);

do
{
    /* Grab program statements one by one and examine them on an individual */
    /* and chronological basis. */
    signal = get_line(&this_line,&comment_line,&info_buffer,infile,error_file);
    search(search_list,delay_data,skeleton,&this_line,flags,&end_list,
           &info_buffer,pkg_name,task_name,messages,procedures,filename,
           error_file);
}
while((signal != 0)&&(flags->finished == NO)&&(flags->fatal_error == NO));

done_parsing;;

/* The critical data structures updated in this procedure are skeleton and */
/* procedures because these hold the growing collection of subprogram and */
/* task models. When task_parse calls parse for the final time to examine */
/* the actual task body, "skeleton" is left with the final task model and */
/* is later passed on to the model analysis procedures */
}
```

Appendix D

```
/*-----  
CALLNAME: CHECK_OVERRUN  
AUTHOR: S. Treadwell  
CREATED: 09 JUN 92  
UPDATED: 20 OCT 92  
  
At this point in the analysis, a worst case path parameterization has been  
established for each task instantiation in the suite. These path  
parameterizations have been used to establish the critical data values for  
each vg, and it is the vg records that are used for the performance failure  
predictions or "overrun checks." This procedure performs the overrun check  
for all 40 vgs (or at least the ones that exist) and leaves the overrun  
info in an array that is part of the vg record. The write_file procedure  
is responsible for later extracting and ciphering the overrun info.  
-----*/  
  
void check_overrun(vg,delay_data,error_file)  
struct vg_info vg[];  
struct constant_list *delay_data;  
FILE *error_file;  
{  
    int calculate_time();  
    int vg_num;  
    int rate_group;  
    int rg_total;  
    int num_tasks[5];  
    int num_tasks_completed[9];  
    int num_msg_queued[8];  
    int num_packets_queued[8];  
    int num_rate_groups_due[8];  
    int overhead[8];  
    int i,j;  
  
    /* For each possible virtual group in the system... */  
    for(vg_num = 0; vg_num < 40; ++vg_num)  
    {  
        if (vg[vg_num].present == YES) /* If the vg has any tasks resident in it */  
        {  
            for(i = 0; i < 5; ++i) /* Initialize intermediate tally variables */  
            {  
                num_tasks[i] = 0;  
                vg[vg_num].rg_total[i] = 0;  
                vg[vg_num].msg_total[i].num_msg_queued = 0;  
                vg[vg_num].msg_total[i].num_packets = 0;  
            }  
  
            for(i = 0; i < vg[vg_num].num_tasks; ++i) /* For each task in the vg...*/  
            {  
                rate_group = vg[vg_num].task[i].rate_group; /* Simplify notation */  
                ++num_tasks[rate_group]; /* Establish how many tasks are in each rg */  
                /* For each rg, establish a tally of delay time for all tasks */  
                vg[vg_num].rg_total[rate_group] +=  
                    vg[vg_num].task[i].counter_set.total_time;  
                /* For each rg, establish a tally of queued packets for all tasks */  
                vg[vg_num].msg_total[rate_group].num_packets +=  
                    vg[vg_num].task[i].counter_set.num_queued;  
                /* For each rg, establish a tally of queued messages for all tasks */  
                vg[vg_num].msg_total[rate_group].num_msg_queued +=
```

Appendix D

```

    vg[vg_num].task[i].counter_set.num_msg_queued;
}

for(i = 0; i < 8; ++i)
    overhead[i] = 0;

/* For each minor frame, establish how many tasks should have      */
/* completed execution cycles during the previous minor frame      */
num_tasks_completed[0] = vg[vg_num].num_tasks;
num_tasks_completed[1] = num_tasks[4];
num_tasks_completed[2] = num_tasks[4] + num_tasks[3];
num_tasks_completed[3] = num_tasks[4];
num_tasks_completed[4] = num_tasks[4] + num_tasks[3] + num_tasks[2];
num_tasks_completed[5] = num_tasks[4];
num_tasks_completed[6] = num_tasks[4] + num_tasks[3];
num_tasks_completed[7] = num_tasks[4];
num_tasks_completed[8] = num_tasks_completed[0];

/* For each minor frame, establish how many messages are queued by  */
/* tasks that should have completed cycles during the previous frame */
num_msg_queued[0] = vg[vg_num].msg_total[4].num_msg_queued +
    vg[vg_num].msg_total[3].num_msg_queued +
    vg[vg_num].msg_total[2].num_msg_queued +
    vg[vg_num].msg_total[1].num_msg_queued;
num_msg_queued[1] = vg[vg_num].msg_total[4].num_msg_queued;
num_msg_queued[2] = vg[vg_num].msg_total[4].num_msg_queued +
    vg[vg_num].msg_total[3].num_msg_queued;
num_msg_queued[3] = num_msg_queued[1];
num_msg_queued[4] = vg[vg_num].msg_total[4].num_msg_queued +
    vg[vg_num].msg_total[3].num_msg_queued +
    vg[vg_num].msg_total[2].num_msg_queued;
num_msg_queued[5] = num_msg_queued[1];
num_msg_queued[6] = num_msg_queued[2];
num_msg_queued[7] = num_msg_queued[1];

/* For each minor frame, establish how many packets are queued by  */
/* tasks that should have completed cycles during the previous frame */
num_packets_queued[0] = vg[vg_num].msg_total[4].num_packets +
    vg[vg_num].msg_total[3].num_packets +
    vg[vg_num].msg_total[2].num_packets +
    vg[vg_num].msg_total[1].num_packets;
num_packets_queued[1] = vg[vg_num].msg_total[4].num_packets;
num_packets_queued[2] = vg[vg_num].msg_total[4].num_packets +
    vg[vg_num].msg_total[3].num_packets;
num_packets_queued[3] = num_packets_queued[1];
num_packets_queued[4] = vg[vg_num].msg_total[4].num_packets +
    vg[vg_num].msg_total[3].num_packets +
    vg[vg_num].msg_total[2].num_packets;
num_packets_queued[5] = num_packets_queued[1];
num_packets_queued[6] = num_packets_queued[2];
num_packets_queued[7] = num_packets_queued[1];

/* For each minor frame, establish how many rate groups should have */
/* encountered frame boundaries at the beginning of the given minor */
/* frame (where the given minor frame * is the index number          */
num_rate_groups_due[0] = 4;
num_rate_groups_due[1] = 1;
num_rate_groups_due[2] = 2;
num_rate_groups_due[3] = 1;

```


Appendix D

```

num_rate_groups_due[4] = 3;
num_rate_groups_due[5] = 1;
num_rate_groups_due[6] = 2;
num_rate_groups_due[7] = 1;

/* Now, for each minor frame, establish the delay due to system      */
/* overhead as a function of the intermediate tally values calculated */
/* above                                                              */
for(i = 0; i < 8; ++i)
{
    /* Delay due to context switches for tasks completing in given frame */
    overhead[i] += num_tasks_completed[i+1] * delay_data->context_switch;

    /* Delay due to scheduling of tasks completing in the previous frame */
    overhead[i] += num_tasks_completed[i] * delay_data->RGD_tsk_coeff;
    overhead[i] += num_rate_groups_due[i] * delay_data->RGD_tsk_const;

    /* Delay due to messages queued by tasks completed in previous frame */
    overhead[i] += num_msg_queued[i] * delay_data->RGD_msg_const;
    overhead[i] += num_packets_queued[i] *
        (delay_data->IH_coeff + delay_data->RGD_msg_coeff);

    overhead[i] += delay_data->local_FDIR;
    overhead[i] += delay_data->IH_const;

    /* Due to the nature of the delay data for the RGD, the following */
    /* distinction is made to account for situations where there are */
    /* no messages queued by the tasks completed in the previous frame */
    if(num_packets_queued[i]>0) overhead[i] += delay_data->RGD_overall_const;
    else overhead[i] += delay_data->RGD_empty_queue_const;
}

/* Check the deadlines for the RG4 tasks */
for(i = 0; i < 8; ++i)
    if ((overhead[i] + vg[vg_num].rg_total[4]) > MINORFRAME)
        vg[vg_num].overrun[i] = YES;

/* Check the deadlines for the RG3 tasks */
rg_total = (2*vg[vg_num].rg_total[4]) + vg[vg_num].rg_total[3];
if((overhead[0]+overhead[1]+rg_total) > (2*MINORFRAME))
    vg[vg_num].overrun[8] = YES;
if((overhead[2]+overhead[3]+rg_total) > (2*MINORFRAME))
    vg[vg_num].overrun[9] = YES;
if((overhead[4]+overhead[5]+rg_total) > (2*MINORFRAME))
    vg[vg_num].overrun[10] = YES;
if((overhead[6]+overhead[7]+rg_total) > (2*MINORFRAME))
    vg[vg_num].overrun[11] = YES;

/* Check the deadlines for the RG2 tasks */
rg_total = (4*vg[vg_num].rg_total[4]) + (2*vg[vg_num].rg_total[3]) +
    vg[vg_num].rg_total[2];
if((overhead[0]+overhead[1]+overhead[2]+overhead[3]+rg_total) >
    (4*MINORFRAME))
    vg[vg_num].overrun[12] = YES;
if((overhead[4]+overhead[5]+overhead[6]+overhead[7]+rg_total) >
    (4*MINORFRAME))
    vg[vg_num].overrun[13] = YES;

/* Check the single deadline for the RG1 tasks */

```

Appendix D

```
rg_total = (8*vg[vg_num].rg_total[4]) + (4*vg[vg_num].rg_total[3]) +
(2*vg[vg_num].rg_total[2]) + vg[vg_num].rg_total[1];
if((overhead[0]+overhead[1]+overhead[2]+overhead[3]+overhead[4]+
overhead[5]+overhead[6]+overhead[7]+rg_total)>(8*MINORFRAME))
vg[vg_num].overrun[14] = YES;

/* Record the overhead data in the vg record */
for(i = 0; i < 8; ++i)
vg[vg_num].overhead[i] = overhead[i];
}
}

/* At the close of this procedure, all results are stored in the vg      */
/* records and sent back to the "main" for interpretation by write_file */
}
```

Appendix D

```
/*-----  
CALLNAME: READ_LIST  
AUTHOR: S. Treadwell  
CREATED: 13 APR 92  
UPDATED: 15 JUN 92  
  
This procedure incorporates info held in external files into the run-time  
data base for the analysis tool. "key_words.dat" holds the critical  
constructs that are the key to parsing the Ada code. "constants.dat" holds  
the system specific delay data that is needed for doing overrun calculations  
and path parameterization comparisons.  
-----*/  
  
void read_list(search_list,delay_data)  
struct list_info *search_list;  
struct constant_list *delay_data;  
{  
    int signal = 0;  
    int value = 0;  
    int num = 0;  
    int k;  
    int dummy_int;  
    char dummy_string[100];  
    FILE *list_file;  
  
    list_file = fopen("key_words.dat","r");  
    if (list_file == NULL)  
    {  
        printf("Cannot open key_words.dat for input\n");  
        exit(2);  
    }  
  
    /* Note that file format has each critical construct on its own line */  
    while(signal != EOF)  
        signal = fscanf(list_file,"%s\n",search_list->entry[num++].name);  
    search_list->length = num - 1;  
    fclose(list_file);  
  
    list_file = fopen("constants.dat","r");  
    if (list_file == NULL)  
    {  
        printf("Cannot open constants.dat for input\n");  
        exit(2);  
    }  
  
    /* File format has a label followed by the related integer value. Each */  
    /* label/value pair occupies its own line */  
    fscanf(list_file,"%s %d\n",dummy_string,&dummy_int);  
    delay_data->queue_coeff = dummy_int;  
    fscanf(list_file,"%s %d\n",dummy_string,&dummy_int);  
    delay_data->queue_const = dummy_int;  
    fscanf(list_file,"%s %d\n",dummy_string,&dummy_int);  
    delay_data->retrieve_coeff = dummy_int;  
    fscanf(list_file,"%s %d\n",dummy_string,&dummy_int);  
    delay_data->retrieve_const = dummy_int;  
    fscanf(list_file,"%s %d\n",dummy_string,&dummy_int);  
    delay_data->IH_coeff = dummy_int;  
    fscanf(list_file,"%s %d\n",dummy_string,&dummy_int);
```

Appendix D

```
delay_data->IH_const = dummy_int;
fscanf(list_file, "%s %d\n", dummy_string, &dummy_int);
delay_data->RGD_msg_coeff = dummy_int;
fscanf(list_file, "%s %d\n", dummy_string, &dummy_int);
delay_data->RGD_msg_const = dummy_int;
fscanf(list_file, "%s %d\n", dummy_string, &dummy_int);
delay_data->RGD_tsk_coeff = dummy_int;
fscanf(list_file, "%s %d\n", dummy_string, &dummy_int);
delay_data->RGD_tsk_const = dummy_int;
fscanf(list_file, "%s %d\n", dummy_string, &dummy_int);
delay_data->RGD_overall_const = dummy_int;
fscanf(list_file, "%s %d\n", dummy_string, &dummy_int);
delay_data->RGD_empty_queue_const = dummy_int;
fscanf(list_file, "%s %d\n", dummy_string, &dummy_int);
delay_data->context_switch = dummy_int;
fscanf(list_file, "%s %d\n", dummy_string, &dummy_int);
delay_data->local_FDIR = dummy_int;
fscanf(list_file, "%s %d\n", dummy_string, &dummy_int);
delay_data->sys_FDIR = dummy_int;

fclose(list_file);
}
```

Appendix D

```
/*-----
CALLNAME: GET_LINE
AUTHOR: S. Treadwell
CREATED: 10 JUN 92
UPDATED: 18 NOV 92

Will use the stream of characters provided by the specified file to assemble
a buffer of single word items all of which belong to a single line of code.
The key here is to look for a semicolon as the line terminator, not a
carriage return. The semicolon is not included in the program statement and
all capital letters are transformed to lower case letters. All white space
is deleted and comment lines are recorded in a separate buffer using the same
format as the program statement. The program statement stored in this_line
is passed back to parse to be examined during model construction. The
comment held in comment_line is passed to parse_comment to extract any
programmer input included.
-----*/

int get_line(this_line,comment_line,info_buffer,infile,error_file)
struct list_info *this_line;
struct list_info *comment_line;
struct comment_info *info_buffer;
FILE *infile;
FILE *error_file;
{
    void parse_comment();
    int k;
    int c = 0;
    int num_word = 0;
    int num_comment = 0;
    int count = 0;
    int signal = 1;
    char temp[80];
    enum boolean comment = NO;
    enum boolean done = NO;

    this_line->length = 0;
    comment_line->length = 0;

    do
    {
        count = 0;
        do
        {
            if ((c == '\n')&&(comment == YES)) /* for "\n" after a comment... */
            {
                comment = NO; /* the comment is terminated by the carriage return */
                comment_line->marker = 0;
                /* Check the completed comment for programmer input */
                parse_comment(comment_line,info_buffer,error_file);
                comment_line->length = 0; /* Once checked,the comment is not needed */
            }
            while(isspace(c = fgetc(infile))); /* kill the white space */
            if (c == EOF) break; /* terminate procedure upon end of file */

        }
        do
        {
            if((c > 64)&&(c < 91)) c += 32; /* transform UPPER case to lower case */
            if((c == ';')&&(comment == NO)) done = YES; /* prog statement complete */
        }
    }
}
```

Appendix D

```
    if(!done) temp[count++] = c; /* add character to word */
}
while(!isspace(c = fgetc(infile)));
if (c == EOF) break;

temp[count] = '\0';
/* add word to line*/
strcpy(this_line->entry[this_line->length].name,temp);
strcpy(comment_line->entry[comment_line->length].name,temp);

if ((k = strncmp(temp,"--",2)) == 0) comment = YES;
if (comment) ++comment_line->length;
else ++this_line->length;
}
while (!done);

if (c == EOF) signal = 0;
this_line->marker = 0;

/* When signal is returned as a "0," it signifies that the end of the file */
/* has been reached. */
return(signal);
}
```

Appendix D

```

/*-----
CALLNAME: SEARCH
AUTHOR: S. Treadwell
CREATED: 11 APR 92
UPDATED: 18 NOV 92

Will do the primary parsing for the "finish" stage. Once a line of code is
available for analysis, 'search' will inspect it for a given list of ada
constructs and system calls and will branch to other parsing and analysis
functions as dictated by what is found. This procedure looks for call names
and ada constructs exactly as given by the user in auxillary files. When
searching for matches, names that are similar but not exactly the same as
the names sought will not be sufficient to warrant a match.
-----*/

void search(search_list, delay_data, skeleton, this_line, flags, end_list,
            info_buffer, pkg_name, task_name, messages, procedures, filename,
            error_file)
struct list_info *search_list;          /* list of constructs          */
struct constant_list *delay_data;      /* list of system specific delay data */
struct model_info *skeleton;           /* holds the current model      */
struct list_info *this_line;          /* current program statement    */
struct flag_list *flags;              /* parsing status info         */
struct list_info *end_list;           /* nesting status info         */
struct comment_info *info_buffer;     /* programmer provided info     */
char pkg_name[];
char task_name[];
struct message_limits messages;       /* message passing limits      */
struct proc_list_info *procedures;    /* holds all subprogram models  */
char filename[];
FILE *error_file;                    /* error log                   */
{
    void process_loop();
    void end_found();
    void task_found();
    void pkg_found();
    void proc_found();
    int find_parameter();
    enum boolean valid_call();
    int i, j, k;
    int old_num_counters;
    int old_length;
    int value;
    int previous;

    for(i = 0; i < this_line->length; ++i) /* check each word in the statement */
    {
        /* establish the index for the word that precedes the word currently
        /* being examined
        if(i > 0) previous = i - 1;
        else previous = 0;

        for(j = 0; j < search_list->length; ++j) /* check against critical const */
        {
            if((k= strcmp(this_line->entry[i].name, search_list->entry[j].name)) == 0)
            switch(j)
            {
                case WFS: /* append to current model */

```

Appendix D

```
skeleton->entry[skeleton->length].type = WFS;
skeleton->entry[skeleton->length].value = UNDEFINED;
if (flags->ctr_active) ++skeleton->length; /* model must be active */
break;
case LOOP:
/* ignore "end loop" statements here */
if ((k = strcmp(this_line->entry[previous].name,"end")) != 0)
{
    this_line->marker = i;
    /* extract extra info and add an entry to the model */
    process_loop(skeleton,end_list,this_line,info_buffer,error_file);
    if (flags->ctr_active) ++skeleton->length;
}
break;
case IF: /* append to current model */
if ((k = strcmp(this_line->entry[previous].name,"end")) != 0)
{
    skeleton->entry[skeleton->length].type = IF;
    skeleton->entry[skeleton->length].value = UNDEFINED;
    if (flags->ctr_active) ++skeleton->length;
    /* update the end_list */
    strcpy(end_list->entry[end_list->length++].name,"if");
}
break;
case ELSE:
skeleton->entry[skeleton->length].type = ELSE;
skeleton->entry[skeleton->length].value = UNDEFINED;
if (flags->ctr_active) ++skeleton->length;
break;
case ELSIF:
skeleton->entry[skeleton->length].type = ELSIF;
skeleton->entry[skeleton->length].value = UNDEFINED;
if (flags->ctr_active) ++skeleton->length;
break;
case CASE: /* ignore "end case" statements here */
if((k = strcmp(this_line->entry[previous].name,"end")) != 0)
{
    flags->enable_when = YES; /* guards against "exception" handling */
    /* code being included in the model */
    skeleton->entry[skeleton->length].type = CASE;
    skeleton->entry[skeleton->length].value = UNDEFINED;
    if (flags->ctr_active) ++skeleton->length;
    /* update the end_list */
    strcpy(end_list->entry[end_list->length++].name,"case");
}
break;
case WHEN:
/* add entry to model only if it belongs to a case statement */
skeleton->entry[skeleton->length].type = WHEN;
skeleton->entry[skeleton->length].value = UNDEFINED;
if ((flags->ctr_active == YES) && (flags->enable_when == YES))
    ++skeleton->length;
break;
case END:
this_line->marker = i + 1;
end_found(delay_data,this_line,end_list,procedures,flags,skeleton,
          pkg_name,task_name,error_file);
break;
case TASK:
```


Appendix D

```

    this_line->marker = i + 1;
    task_found(this_line,end_list,task_name,flags,error_file);
    break;
case PACKAGE:
    this_line->marker = i + 1;
    pkg_found(this_line,end_list,pkg_name,flags,error_file);
    break;
case PROC:
    this_line->marker = i;
    proc_found(this_line,end_list,procedures,flags,filename,error_file);
    break;
case BEGIN:
    if ((flags->proc_depth > 0)|| (flags->task_found == YES))
        flags->ctr_active = YES;
    break;
case SELECT:
    if((k = strcmp(this_line->entry[previous].name,"end")) != 0)
        strcpy(end_list->entry[end_list->length++].name,"select");
    break;
case RECORD:
    if((k = strcmp(this_line->entry[previous].name,"end")) != 0)
        strcpy(end_list->entry[end_list->length++].name,"record");
    break;
case ACCEPT:
    strcpy(end_list->entry[end_list->length++].name,"start");
    break;
default: break;
}

/* Need a separate comparison statement to deal specifically with the */
/* message passing functions because there could be a ( and parameter */
/* names attached to the function call which would cause an inexact */
/* match with the key word in search_list */
if((k= strncmp(this_line->entry[i].name,search_list->entry[j].name,
               strlen(search_list->entry[j].name))) == 0)
{
    this_line->marker = i;
    search_list->marker = j;
    switch(j)
    {
    case QUEUE:
        /* check to make sure it really is a subprogram call */
        if(valid_call(this_line->entry[i].name,
                     search_list->entry[j].name)==YES)
        {
            skeleton->entry[skeleton->length].type = QUEUE;
            /* If programmer input specifies message size, use that value */
            /* over any other possible size values */
            value = info_buffer->message_size;
            info_buffer->message_size = DEFAULT;
            if(value == DEFAULT)
                /* If programmer input does not specify size, find the value */
                value = find_parameter(search_list,this_line,error_file);
            /* If no viable value can be established, assume the default */
            if((value == LOST)|| (value > messages.xmit_size))
            {
                fprintf(error_file,"Assuming default size for queue_message\n");
                skeleton->entry[skeleton->length].value =
                    packetize(messages.xmit_size);
            }
        }
    }
}

```

Appendix D

```

    }
    else
        skeleton->entry[skeleton->length].value =
            packetize(value); /* transform size from bytes to packets */
    if (flags->ctr_active) ++skeleton->length;
}
break;
case RETRIEVE:
    if(valid_call(this_line->entry[i].name,
                 search_list->entry[j].name)==YES)
    {
        skeleton->entry[skeleton->length].type = RETRIEVE;
        value = info_buffer->message_size;
        info_buffer->message_size = DEFAULT;
        if(value == DEFAULT)
            value = find_parameter(search_list,this_line,error_file);
        if((value == LOST)|| (value > messages.xmit_size))
        {
            fprintf(error_file,
                   "Assuming default size for retrieve_message\n");
            skeleton->entry[skeleton->length].value =
                packetize(messages.xmit_size);
        }
    }
    else
        skeleton->entry[skeleton->length].value =
            packetize(value);
    if (flags->ctr_active) ++skeleton->length;
}
break;
case SEND:
    if(valid_call(this_line->entry[i].name,
                 search_list->entry[j].name)==YES)
    {
        skeleton->entry[skeleton->length].type = SEND;
        value = info_buffer->message_size;
        info_buffer->message_size = DEFAULT;
        if(value == DEFAULT)
            value = find_parameter(search_list,this_line,error_file);
        if((value == LOST)|| (value > messages.xmit_size))
        {
            fprintf(error_file,"Assuming default size for send_message\n");
            skeleton->entry[skeleton->length].value =
                packetize(messages.xmit_size);
        }
    }
    else
        skeleton->entry[skeleton->length].value =
            packetize(value);
    if (flags->ctr_active) ++skeleton->length;
}
break;
case READ:
    if(valid_call(this_line->entry[i].name,
                 search_list->entry[j].name)==YES)
    {
        skeleton->entry[skeleton->length].type = READ;
        value = info_buffer->message_size;
        info_buffer->message_size = DEFAULT;
        if(value == DEFAULT)
            value = find_parameter(search_list,this_line,error_file);
    }
}

```

Appendix D

```

    if((value == LOST)|| (value > messages.xmit_size))
    {
        fprintf(error_file,"Assuming default size for read_message\n");
        skeleton->entry[skeleton->length].value =
            packetize(messages.xmit_size);
    }
    else
        skeleton->entry[skeleton->length].value =
            packetize(value);
    if (flags->ctr_active) ++skeleton->length;
}
break;
default: break;
}
}
}

/* Now look for subprogram calls and insert models as appropriate */
if (flags->ctr_active)
for(j = 0; j < procedures->length; ++j)
    if((k = strncmp(this_line->entry[i].name,procedures->entry[j].name,
        strlen(procedures->entry[j].name))) == 0)
    {
        old_num_counters = skeleton->num_counters;
        old_length = skeleton->length;
        for(k = 0; k < procedures->entry[j].skeleton.length; ++k)
            skeleton->entry[skeleton->length++] =
                procedures->entry[j].skeleton.entry[k];
        for(k = 0; k < procedures->entry[j].skeleton.num_counters; ++k)
            skeleton->counter_set[skeleton->num_counters++] =
                procedures->entry[j].skeleton.counter_set[k];
        /* adjust counter_set index values to accomodate the new entries */
        for(k = old_length; k < skeleton->length; ++k)
            if(skeleton->entry[k].type == COUNTERSET)
                skeleton->entry[k].value += old_num_counters;
    }
}
}
}

```

Appendix D

```

/*-----
CALLNAME: PARSE_COMMENT
AUTHOR: S. Treadwell
CREATED: 12 MAR 93
UPDATED: 14 MAR 93

This procedure takes a single complete comment as input and extracts from
it any programmer input specifications included. The specs must conform to
the established conventions, and any information found is used to update the
info_buffer, which is a running list of default values to use as loop
iteration maximums and message sizes.
-----*/
void parse_comment(comment_line, info_buffer, error_file)
struct list_info *comment_line;
struct comment_info *info_buffer;
FILE *error_file;
{
    int evaluate();
    int eval_simple_num();
    struct string key_word[5];
    int i, j, k;
    int value = UNDEFINED;
    strcpy(key_word[0].name, "basic");
    strcpy(key_word[1].name, "for");
    strcpy(key_word[2].name, "while");
    strcpy(key_word[3].name, "message");
    /* Check if comment line conforms to programmer input conventions */
    if((k = strcmp(comment_line->entry[1].name, "*")) == 0)
    {
        /* if an integer value is given in the comment, evaluate it */
        if(evaluate(comment_line->entry[comment_line->length-1].name) == SIMPLENUM)
            value = eval_simple_num(comment_line->entry[comment_line->length-1].name);
        for(i = 0; i < 4; ++i)
            if((k = strncmp(comment_line->entry[2].name, key_word[i].name,
                strlen(key_word[i].name)-1)) == 0)
                /* update the appropriate info buffer value */
                /* if value is not a positive integer, the default value is assumed */
                switch(i)
                {
                    case 0:
                        if (value < 0) value = INFINITE;
                        info_buffer->basic_loop_limit = value;
                        break;
                    case 1:
                        if (value < 0) value = UNDEFINED;
                        info_buffer->for_loop_limit = value;
                        break;
                    case 2:
                        if (value < 0) value = UNDEFINED;
                        info_buffer->while_loop_limit = value;
                        break;
                    case 3:
                        if (value < 0) value = DEFAULT;
                        info_buffer->message_size = value;
                        break;
                    default: break;
                }
    }
}
}

```

Appendix D

```
/*-----  
CALLNAME: WRITE_FILE  
AUTHOR: S. Treadwell  
CREATED: 05 APR 93  
UPDATED: 28 APR 93  
  
This procedure sends the analysis results to an external file called  
"results.dat." All information is passed in through the ug data structure.  
The output format is self-explanatory.  
-----*/
```

```
void write_file(ug)  
struct ug_info ug[];  
{  
    int i,j,k;  
    int left_over;  
    int RG3_left_over;  
    int RG2_left_over;  
    int RG1_left_over;  
    FILE *outfile;  
  
    outfile = fopen("results.dat","w");  
    if(outfile == NULL)  
    {  
        printf("Cannot open results.dat for output\n");  
        exit(2);  
    }  
  
    fprintf(outfile,"All results are given in terms of microseconds.\n");  
    fprintf(outfile,"The allotted minor frame time is %d.\n\n",MINORFRAME);  
  
    for(i = 0; i < 40; ++i)  
    {  
        if(ug[i].present == YES)  
        {  
            fprintf(outfile,"RESULTS for UG*%d\n",i);  
            /* List individual results for tasks */  
            for(j = 0; j < ug[i].num_tasks; ++j)  
            {  
                fprintf(outfile,"\nTASK: %s    RATE GROUP: %d\n",ug[i].task[j].name,  
                    ug[i].task[j].rate_group);  
                fprintf(outfile,"WORST CASE PATH: number of packets queued: %d\n",  
                    ug[i].task[j].counter_set.num_queued);  
                fprintf(outfile,"                    number of messages queued: %d\n",  
                    ug[i].task[j].counter_set.num_msg_queued);  
                fprintf(outfile,"                    number of packets retrieved: %d\n",  
                    ug[i].task[j].counter_set.num_retrieved);  
                fprintf(outfile,"                    number of messages retrieved: %d\n",  
                    ug[i].task[j].counter_set.msg_retrieved);  
                fprintf(outfile,"                    number of packets sent: %d\n",  
                    ug[i].task[j].counter_set.num_sent);  
                fprintf(outfile,"                    number of packets read: %d\n",  
                    ug[i].task[j].counter_set.num_read);  
                fprintf(outfile,"                    minimal delay: %d\n",  
                    ug[i].task[j].counter_set.total_time);  
            }  
            /* Now categorize results according to rate groups and frame # */  
            fprintf(outfile,"\nRATE GROUP TOTALS FOR APPLICATION TASKS\n");
```

Appendix D

```

for(j = 1; j < 5; ++j)
    fprintf(outfile,"rate group %d: %d\n",j,vg[i].rg_total[j]);
fprintf(outfile,"\nOVERHEAD TOTALS\n");

/* Create a matrix of results to show the contribution from overhead */
/* and each rate group for each minor frame. */
for(j = 0; j < 8; ++j)
    fprintf(outfile,"minor frame %d: %d\n",j,vg[i].overhead[j]);
fprintf(outfile,
        "\nMINOR FRAME OVERHEAD    RG4    RG3    RG2    RG1\n");
RG3_left_over = vg[i].rg_total[3];
RG2_left_over = vg[i].rg_total[2];
RG1_left_over = vg[i].rg_total[1];
for(j = 0; j < 8; ++j)
{
    /* Determine whether a new frame begins for RG3 and RG2 tasks */
    if((j == 2)||j == 4)||j == 6)
        RG3_left_over = vg[i].rg_total[3];
    if(j == 4)
        RG2_left_over = vg[i].rg_total[2];
    fprintf(outfile,"    %d    ",j);
    /* overhead contribution */
    if(MINORFRAME >= vg[i].overhead[j])
    {
        fprintf(outfile,"    %5d",vg[i].overhead[j]);
        left_over = MINORFRAME - vg[i].overhead[j];
    }
    else
    {
        fprintf(outfile,"    %5d",MINORFRAME);
        left_over = 0;
    }
    /* RG4 contribution */
    if(left_over >= vg[i].rg_total[4])
    {
        fprintf(outfile,"    %5d",vg[i].rg_total[4]);
        left_over -= vg[i].rg_total[4];
    }
    else
    {
        fprintf(outfile,"    %5d",left_over);
        left_over = 0;
    }
    /* RG3 contribution */
    if(left_over >= RG3_left_over)
    {
        fprintf(outfile,"    %5d",RG3_left_over);
        left_over -= RG3_left_over;
        RG3_left_over = 0;
    }
    else
    {
        fprintf(outfile,"    %5d",left_over);
        RG3_left_over -= left_over;
        left_over = 0;
    }
    /* RG2 contribution */
    if(left_over >= RG2_left_over)
    {

```

Appendix D

```

    fprintf(outfile, "  %5d", RG2_left_over);
    left_over -= RG2_left_over;
    RG2_left_over = 0;
}
else
{
    fprintf(outfile, "  %5d", left_over);
    RG2_left_over -= left_over;
    left_over = 0;
}
/* RG1 contribution */
if(left_over >= RG1_left_over)
{
    fprintf(outfile, "  %5d\n", RG1_left_over);
    left_over -= RG1_left_over;
    RG1_left_over = 0;
}
else
{
    fprintf(outfile, "  %5d\n", left_over);
    RG1_left_over -= left_over;
    left_over = 0;
}
}
/* State predicted overrun conditions */
fprintf(outfile, "\n");
if(vg[i].overrun[0] == YES)
    fprintf(outfile, "RG4 did not satisfy its boundary in frame 0.\n");
if(vg[i].overrun[1] == YES)
    fprintf(outfile, "RG4 did not satisfy its boundary in frame 1.\n");
if(vg[i].overrun[2] == YES)
    fprintf(outfile, "RG4 did not satisfy its boundary in frame 2.\n");
if(vg[i].overrun[3] == YES)
    fprintf(outfile, "RG4 did not satisfy its boundary in frame 3.\n");
if(vg[i].overrun[4] == YES)
    fprintf(outfile, "RG4 did not satisfy its boundary in frame 4.\n");
if(vg[i].overrun[5] == YES)
    fprintf(outfile, "RG4 did not satisfy its boundary in frame 5.\n");
if(vg[i].overrun[6] == YES)
    fprintf(outfile, "RG4 did not satisfy its boundary in frame 6.\n");
if(vg[i].overrun[7] == YES)
    fprintf(outfile, "RG4 did not satisfy its boundary in frame 7.\n");
if(vg[i].overrun[8] == YES)
    fprintf(outfile, "RG3 did not satisfy its boundary in frame 1.\n");
if(vg[i].overrun[9] == YES)
    fprintf(outfile, "RG3 did not satisfy its boundary in frame 3.\n");
if(vg[i].overrun[10] == YES)
    fprintf(outfile, "RG3 did not satisfy its boundary in frame 5.\n");
if(vg[i].overrun[11] == YES)
    fprintf(outfile, "RG3 did not satisfy its boundary in frame 7.\n");
if(vg[i].overrun[12] == YES)
    fprintf(outfile, "RG2 did not satisfy its boundary in frame 3.\n");
if(vg[i].overrun[13] == YES)
    fprintf(outfile, "RG2 did not satisfy its boundary in frame 7.\n");
if(vg[i].overrun[14] == YES)
    fprintf(outfile, "RG1 did not satisfy its only boundary.\n");
fprintf(outfile, "\f");
}
}

```

Appendix D

```
/*-----  
CALLNAME: WITH_FOUND  
AUTHOR: S. Treadwell  
CREATED: 12 JUL 92  
UPDATED: 08 OCT 92  
  
This procedure is used to help establish the software hierarchy for a given  
task. Whenever a "with" statement is found in a context clause, it is sent  
to this procedure so that the package name can be extracted and added to  
pkg_list.  
-----*/  
  
void with_found(this_line, pkg_list)  
struct list_info *this_line;  
struct list_info *pkg_list;  
{  
    int i, c;  
    int j = 0;  
    int k = 0;  
    int num = 0;  
    enum boolean repeat = NO;  
    char temp1[80];  
    char temp2[80];  
    struct string package[10];  
  
    /* Most of the complexity of this procedure is due to an attempt to handle */  
    /* different format styles for the "with" statement -- i.e. single vs.    */  
    /* multiple names on each line, with or without white space separating the */  
    /* names.                                                                    */  
    for(i = this_line->marker; i < this_line->length; ++i)  
    {  
        strcpy(temp1, this_line->entry[i].name);  
        while((c = temp1[j++]) != '\0')  
        {  
            if(c == ",")  
            {  
                temp2[k] = '\0';  
                if ((c = temp1[j]) != '\0')  
                {  
                    k = 0;  
                    if (strlen(temp2) >= 1) strcpy(package[num++].name, temp2);  
                }  
            }  
            else  
                temp2[k++] = c;  
        }  
        temp2[k] = '\0';  
        if (strlen(temp2) >= 1) strcpy(package[num++].name, temp2);  
        j = 0;  
        k = 0;  
    }  
  
    /* Ensure that package names are not repeated within pkg_list */  
    for(i = 0; i < num; ++i)  
    {  
        repeat = NO;  
        for(j = 0; j < pkg_list->length; ++j)  
            if((k = strcmp(package[i].name, pkg_list->entry[j].name)) == 0)
```


Appendix D

```
        repeat = YES;
    if(repeat == NO)
        strcpy(pkg_list->entry[pkg_list->length++].name,package[i].name);
    }
}

/*-----*/
CALLNAME: PKG_FOUND
AUTHOR: S. Treadwell
CREATED: 06 OCT 92
UPDATED: 06 NOV 92

In the case where a package statement is found, this procedure will decide
whether it is a valid package body and ensure that the name matches
the name expected.  If these conditions pass, the proper flag will be set.
-----*/

void pkg_found(this_line,end_list,pkg_name,flags,error_file)
struct list_info *this_line;
struct list_info *end_list;
char pkg_name[];
struct flag_list *flags;
FILE *error_file;
{
    int k;

    if ((k = strcmp(this_line->entry[this_line->marker++].name,"body")) == 0)
    {
        /* In the case where the task body file is being processed the package */
        /* name is not known prior to processing and is defined as "none"      */
        /* When the package name is found it is appropriately recorded          */
        if ((k = strcmp(pkg_name,"none")) == 0)
            strcpy(pkg_name,this_line->entry[this_line->marker].name);
        if ((k = strcmp(pkg_name,this_line->entry[this_line->marker].name)) == 0)
        {
            flags->pkg_found = YES;
            strcpy(end_list->entry[end_list->length++].name,pkg_name);
        }
        else
            fprintf(error_file,"Unexpected package name or instance found\n");
    }
}
```

Appendix D

```
/*-----  
CALLNAME: TASK_FOUND  
AUTHOR: S. Treadwell  
CREATED: 06 OCT 92  
UPDATED: 12 NOV 92  
  
In the case where a task statement is found, this procedure will decide  
whether it is a task body and ensure that the name matches  
the name expected. If these conditions pass, the proper flag will be set.  
-----*/
```

```
void task_found(this_line, end_list, task_name, flags, error_file)  
struct list_info *this_line;  
struct list_info *end_list;  
char task_name[];  
struct flag_list *flags;  
FILE *error_file;  
{  
    int k;  
  
    if ((k = strcmp(this_line->entry[this_line->marker++].name, "body")) == 0)  
    {  
        strcpy(end_list->entry[end_list->length++].name,  
            this_line->entry[this_line->marker].name);  
        if ((k = strcmp(this_line->entry[this_line->marker].name, task_name)) == 0)  
            if (flags->pkg_found == YES) flags->task_found = YES;  
            else fprintf(error_file, "Task found but package not yet found\n");  
        else  
            fprintf(error_file, "Unexpected task name -- found: %s expected: %s\n",  
                this_line->entry[this_line->marker].name, task_name);  
    }  
}
```

```
/*-----  
CALLNAME: PACKETIZE  
AUTHOR: S. Treadwell  
CREATED: 20 APR 93  
UPDATED: 20 APR 93  
  
A simple function to transform a message size in bytes to a message size in  
64-byte packets. Note that four bytes of overhead are added to the message  
size to account for message header information.  
-----*/
```

```
int packetize(num_bytes)  
int num_bytes;  
{  
    int num_packets;  
  
    num_bytes += 4;  
    num_packets = ceil(num_bytes/64.0);  
  
    return(num_packets);  
}
```

Appendix D

```
/*-----  
CALLNAME: PROC_FOUND  
AUTHOR: S. Treadwell  
CREATED: 15 JUL 92  
UPDATED: 10 NOV 92  
  
Grabs the name of a procedure and increments both the length of the procedure  
list and the internal procedure marker.  
-----*/  
  
void proc_found(this_line,end_list,procedures,flags,filename,error_file)  
struct list_info *this_line;  
struct list_info *end_list;  
struct proc_list_info *procedures;  
struct flag_list *flags;  
char filename[];  
FILE *error_file;  
{  
    int i = 0;  
    int c,k;  
    char temp[80];  
    char name[40];  
  
    /* The proc_depth flag is needed to help identify valid subprogram body */  
    /* code so that extraneous code is not included in the model. Basically */  
    /* when proc_depth is greater than zero, we are dealing with code inside */  
    /* a subprogram body and can add model entries appropriately */  
    if(flags->pkg_found == YES) ++flags->proc_depth;  
  
    /* Extract the name of the procedure and strip away the parameter list */  
    strcpy(temp,this_line->entry[this_line->marker + 1].name);  
    c = temp[0];  
    while((c != '\0') && (c != '('))  
    {  
        name[i++] = c;  
        c = temp[i];  
    }  
    name[i] = '\0';  
  
    /* Set up the procedures data structure to accept model info */  
    strcpy(end_list->entry[end_list->length++].name,name);  
    strcpy(procedures->entry[procedures->length].name,name);  
    strcpy(procedures->entry[procedures->length].filename,filename);  
    procedures->entry[procedures->length].done == NO;  
    procedures->marker = procedures->length;  
    ++procedures->length;  
}
```

Appendix D

```
/*-----  
CALLNAME: FIND_PARAMETER  
AUTHOR: S. Treadwell  
CREATED: 4 MAY 92  
UPDATED: 30 OCT 92  
  
Once a system call is found, it may be necessary to calculate the delay it  
incurs based upon the value of one of the parameters included in the call.  
This procedure will identify all of the parameters for a given call, and  
the critical parameter can be isolated and evaluated with reference to all  
preceding code. The object is to attach a specific value to the critical  
parameter so that this value can be used in calculating the expected time  
delay for a given system call. For now, this procedure is only utilized  
when trying to determine the size of a message for a message passing fcn.  
-----*/
```

```
int find_parameter(search_list, this_line, error_file)  
struct list_info *search_list;  
struct list_info *this_line;  
FILE *error_file;  
{  
    int evaluate();  
    int eval_simple_num();  
    int eval_complex_num();  
    int eval_natural_num();  
    int c;  
    int i = 0;  
    int j = 0;  
    int k = 0;  
    int num = 0;  
    int value = LOST;  
    enum boolean foundfirst = NO;  
    enum boolean foundsecond = NO;  
    enum boolean innerset = NO;  
    char temp1[40];  
    char temp2[40];  
    char critical[40];  
    struct string parameter[10];  
  
    /* Most of the complexity of this procedure is due to an attempt to handle */  
    /* any possible parameter list format -- to include white space and even */  
    /* carriage returns between parameters. */  
    for(i = this_line->marker; i < this_line->length; ++i)  
    {  
        strcpy(temp1, this_line->entry[i].name);  
        while((c = temp1[j++]) != '\0')  
        {  
            switch(c)  
            {  
                case '(': /* Looking for the ( that begins the parameter list */  
                    if (foundfirst == YES) /* in case of parameter like "natural(i)" */  
                    {  
                        innerset = YES;  
                        temp2[k++] = c;  
                    }  
                    else foundfirst = YES;  
                    break;  
                case ',': /* commas separate the parameter items */
```

Appendix D

```

    temp2[k] = '\0';
    strcpy(parameter[num].name,temp2);
    k = 0;
    ++num;
    break;
case ')': /* Looking for the closing ) */
    if (innerset == YES)
    {
        temp2[k++] = c;
        innerset = NO;
    }
    else
    {
        foundsecond = YES;
        temp2[k] = '\0';
        strcpy(parameter[num].name,temp2);
    }
    break;
default:
    if (foundfirst == YES)
        if (!isspace(c))
            temp2[k++] = c;
    break;
}
}
j = 0;
}

/* Note that the fourth parameter is targeted because it is the fourth */
/* parameter that specifies message size for message passing functions */
/* In the future, we can pass in the number of the critical parameter */
/* rather than having it hardwired to = 4 */
if (num >= 3)
    strcpy(critical,parameter[3].name);
else
    fprintf(error_file,"Found too few parameters for an instance of %s\n",
        search_list->entry[search_list->marker].name);

/* Determine the format of the critical parameter and evaluate if possible */
switch(evaluate(critical))
{
case SIMPLENUM: value = eval_simple_num(critical); break;
case COMPLEXNUM: value = eval_complex_num(critical); break;
case NATURALNUM: value = eval_natural_num(critical); break;
case UARNAME: value = LOST; break;
case UNKNOWN: value = LOST; break;
}

if (value < 0) value = LOST;

return(value);
}

```

Appendix D

```
/*-----  
CALLNAME: PROCESS_LOOP  
AUTHOR: S. Treadwell  
CREATED: 23 JUL 92  
UPDATED: 18 NOV 92  
  
Will take a "loop" occurrence and further define it into the type of loop or  
classify it as an "end loop." If the program statement defines the beginning  
of a loop, the value of the loop iterations maximum must be defined and  
included in the appropriate model entry.  
-----*/
```

```
void process_loop(skeleton,end_list,this_line,info_buffer,error_file)  
struct model_info *skeleton;  
struct list_info *this_line;  
struct list_info *end_list;  
struct comment_info *info_buffer;  
FILE *error_file;  
{  
    int for_loop_found();  
    enum boolean target_found();  
    int k;  
  
    /* Update end_list according to the loop construct */  
    strcpy(end_list->entry[end_list->length++].name,"loop");  
    skeleton->entry[skeleton->length].value = UNDEFINED;  
  
    /* Is it a for..loop? */  
    if (target_found(this_line,"for","loop") == YES)  
    {  
        skeleton->entry[skeleton->length].type = FOR_LOOP;  
        skeleton->entry[skeleton->length].value =  
            for_loop_found(this_line,info_buffer,error_file);  
    }  
    else /* Is it a while..loop? */  
    {  
        if (target_found(this_line,"while","loop") == YES)  
        {  
            skeleton->entry[skeleton->length].type = WHILE_LOOP;  
            skeleton->entry[skeleton->length].value = info_buffer->while_loop_limit;  
            info_buffer->while_loop_limit = UNDEFINED;  
        }  
        else /* we know it is a basic loop at this point */  
        {  
            skeleton->entry[skeleton->length].type = LOOP;  
            skeleton->entry[skeleton->length].value = info_buffer->basic_loop_limit;  
            info_buffer->basic_loop_limit = INFINITE;  
        }  
    }  
}
```

Appendix D

```
/*-----  
CALLNAME: FOR_LOOP_FOUND  
AUTHOR: S. Treadwell  
CREATED: 4 MAY 92  
UPDATED: 29 JUL 92  
  
If a for..loop construct is found, the loop's max iterations must be defined  
and stored in the value element of the model. This procedure checks the  
programmer input buffer, and if no limit is given there, it finds the loop  
argument and attempts to evaluate it.  
-----*/  
  
int for_loop_found(this_line,info_buffer,error_file)  
struct list_info *this_line;  
struct comment_info *info_buffer;  
{  
    int evaluate();  
    int eval_range_num();  
    struct range_info temprange;  
    char argument[20];  
    int loop_limit = UNDEFINED;  
  
    /* Identify the loop argument */  
    strcpy(argument,this_line->entry[this_line->marker + 3].name);  
  
    /* If there is no programmer input for the loop maximum evaluate the */  
    /* argument if possible */  
    if (info_buffer->for_loop_limit == UNDEFINED)  
        if (evaluate(argument)== RANGENUM)  
            loop_limit = eval_range_num(argument,error_file);  
  
    if(loop_limit < 0) /* If no positive integer is established for limit */  
    {  
        loop_limit = info_buffer->for_loop_limit;  
        info_buffer->for_loop_limit = UNDEFINED;  
    }  
  
    return(loop_limit);  
}
```

Appendix D

```

/*-----
CALLNAME: END_FOUND
AUTHOR: S. Treadwell
CREATED: 15 JUL 92
UPDATED: 18 NOV 92

This procedure is used for any "end" statement found and it is essential to
maintaining the end_list, which holds code nesting status info. For "end"
statements applying to program units, this procedure initiates the model
wrap-up and processing.
-----*/

void end_found(delay_data,this_line,end_list,procedures,flags,skeleton,
              pkg_name,task_name,error_file)
struct constant_list *delay_data;
struct list_info *this_line;
struct list_info *end_list;
struct proc_list_info *procedures;
struct flag_list *flags;
struct model_info *skeleton;
char pkg_name[];
char task_name[];
FILE *error_file;
{
    void reduce_model();
    int i,j,k;
    char dummy_name[80];
    char object[80];
    char expected[80];

    i = procedures->marker;

    strcpy(expected,end_list->entry[end_list->length - 1].name);

    /* If "end" is the last word in the program statement...*/
    if(this_line->marker < this_line->length)
        strcpy(object,this_line->entry[this_line->marker].name); /**/
    else
        strcpy(object,"none");

    /* Compare what end_list expects with what is found in the code. If there */
    /* is a mismatch, declare a fatal error. */
    if((k = strcmp(object,end_list->entry[end_list->length - 1].name)) != 0)
        if((k = strcmp(object,"none")) != 0)/* don't compare something to nothing*/
        {
            flags->fatal_error = YES;
            fprintf(error_file,"Fatal error occurred in package %s with 'end %s'",
                pkg_name,object);
            fprintf(error_file,"...expecting end %s\n",
                end_list->entry[end_list->length - 1].name);
        }

    /* If the end of a subprogram body is found... */
    if((k = strcmp(expected,procedures->entry[i].name)) == 0)
    {
        if(flags->proc_depth > 0) --flags->proc_depth;
        else fprintf(error_file,"Error with procedure depth\n");
        flags->ctr_active = NO; /* deactivate the model */
    }
}

```


Appendix D

```
reduce_model(skeleton,delay_data,error_file); /* wrap-up the model */
procedures->entry[i].skeleton = *skeleton; /* store the model */
procedures->entry[i].done = YES;
skeleton->length = 0; skeleton->num_counters = 0;
procedures->marker = procedures->length;
/* Must account for nested procedure bodies */
for (j = procedures->marker - 1; j >=0; --j)
{
    if(procedures->entry[j].done == NO)
    {
        procedures->marker = j;
        break;
    }
}
}

/* If the end of a package is found... */
if((k = strcmp(object, pkg_name)) == 0)
{
    flags->ctr_active = NO; /* deactivate model */
    flags->finished = YES; /* code processing is complete */
    for(j = 0; j < procedures->length; ++j)
        procedures->entry[j].done = YES;
    /* adjust subprogram names to include package name */
    for(j = procedures->pkg_marker; j < procedures->length; ++j)
    {
        strcpy(dummy_name, pkg_name);
        strcat(dummy_name, ".");
        strcat(dummy_name, procedures->entry[j].name);
        strcpy(procedures->entry[j].name, dummy_name);
    }
    procedures->pkg_marker = procedures->length;
}

/* If the end of a task body is found... */
if((k = strcmp(object, task_name)) == 0)
{
    flags->ctr_active = NO;
    flags->finished = YES;
    for(j = 0; j < procedures->length; ++j)
        procedures->entry[j].done = YES;
}

/* Add an end_loop entry to the model */
if((k = strcmp(expected, "loop")) == 0)
{
    if((k = strcmp(object, "none")) == 0)
        fprintf(error_file, "Fatal error -- expecting an end loop\n");
    skeleton->entry[skeleton->length].type = END_LOOP;
    skeleton->entry[skeleton->length].value = UNDEFINED;
    if (flags->ctr_active) ++skeleton->length;
}

/* Add an end_if entry to the model */
if((k = strcmp(expected, "if")) == 0)
{
    if((k = strcmp(object, "none")) == 0)
        fprintf(error_file, "Fatal error -- expecting an end if\n");
    skeleton->entry[skeleton->length].type = END_IF;
}
```

Appendix D

```
skeleton->entry[skeleton->length].value = UNDEFINED;
if (flags->ctr_active) ++skeleton->length;
}

/* Add an end_case entry to the model */
if((k = strcmp(expected,"case")) == 0)
{
    if((k = strcmp(object,"none")) == 0)
        fprintf(error_file,"Fatal error -- expecting an end case\n");
    skeleton->entry[skeleton->length].type = END_CASE;
    skeleton->entry[skeleton->length].value = UNDEFINED;
    if (flags->ctr_active) ++skeleton->length;
    flags->enable_when = NO;
}

if((k = strcmp(expected,"start")) == 0)
    if((k = strcmp(object,"none")) == 0)
        fprintf(error_file,"Fatal error -- expecting an end start\n");

if((k = strcmp(expected,"select")) == 0)
    if((k = strcmp(object,"none")) == 0)
        fprintf(error_file,"Fatal error -- expecting an end select\n");

if((k = strcmp(expected,"record")) == 0)
    if((k = strcmp(object,"none")) == 0)
        fprintf(error_file,"Fatal error -- expecting an end record\n");

if(end_list->length > 0) --end_list->length;
}
```

Appendix D

```
/*-----  
CALLNAME: VALID_CALL  
AUTHOR: S. Treadwell  
CREATED: 22 JUN 92  
UPDATED: 22 JUN 92  
  
A quick check to see if a procedure call instance is really a valid one.  
Basically this traps for cases where there is a larger word that contains  
the name of a procedure but is not a call for that procedure.  
-----*/
```

```
enum boolean valid_call(argument,match)  
char argument[];  
char match[];  
{  
    int k;  
    enum boolean flag = NO;  
  
    if(strlen(argument) == strlen(match)) flag = YES;  
    if((k = argument[strlen(match)]) == '(') flag = YES;  
  
    return(flag);  
}
```

```
/*-----  
CALLNAME: PRINT_LINE  
AUTHOR: S. Treadwell  
CREATED: 13 APR 92  
UPDATED: 28 OCT 92  
  
Used for debugging purposes. Will print out whatever line of code has been  
most recently stored in the this_line buffer.  
-----*/
```

```
void print_line(this_line,outfile)  
struct list_info *this_line;  
FILE *outfile;  
{  
    int i;  
  
    for (i = 0; i < this_line->length; ++i)  
        fprintf(outfile,"%s ",this_line->entry[i].name);  
    if(this_line->length > 0) fprintf(outfile,"\n");  
}
```

Appendix D

```
/*-----  
CALLNAME: TARGET_FOUND  
AUTHOR: S. Treadwell  
CREATED: 29 JUL 92  
UPDATED: 29 JUL 92  
  
Used in parsing loop statements. Helpful in distinguishing between a for_  
loop and a while_loop since loops are keyed on the word "loop" and not on  
"for" or "while." The boundary string is used to bound the search for the  
beginning of the loop. We don't want to search infinitely and possibly find  
a loop beginning that does not apply to the given loop. Consider the  
following example:  
  for i in 1..10 loop  
    loop  
    ...  
  end loop;  
end loop;  
When parsing the inner loop construct, we don't want to find the beginning  
of the outer loop and mistakenly identify a basic loop as a for..loop.  
-----*/  
  
enum boolean target_found(this_line,target,boundary)  
struct list_info *this_line;  
char target[];  
char boundary[];  
{  
  int i,k;  
  enum boolean success = NO;  
  
  /* Start at the word "loop" and search backwards for the target without */  
  /* going further back than allowed by the boundary set. */  
  for(i = this_line->marker - 1; i >= 0; --i)  
  {  
    if((k = strcmp(this_line->entry[i].name,target)) == 0)  
    {  
      success = YES;  
      this_line->marker = i;  
      break;  
    }  
    if((k = strcmp(this_line->entry[i].name,boundary)) == 0)  
      break;  
  }  
  
  /* The success integer indicates whether the target was found */  
  return(success);  
}
```

Appendix D

```
/*-----  
CALLNAME: EVAL_RANGE_NUM  
AUTHOR: S. Treadwell  
CREATED: 5 MAY 92  
UPDATED: 15 JUN 92  
  
Will take a string expression for a range like '0..60' and parse it into a  
first, last, and span values. These values are then stored in the data  
structure called range, and the span value is also returned as the output of  
the function.  
-----*/
```

```
int eval_range_num(word,error_file)  
char word[];  
{  
    int evaluate();  
    int eval_simple_num();  
    int eval_natural_num();  
    int eval_complex_num();  
    int c;  
    int i = 0;  
    int j = 0;  
    char min[20];  
    char max[20];  
    struct range_info temprange;  
  
    strcpy(temprange.description,word);  
  
    /* find the first value in the range */  
    while(isalnum(c = word[i]))  
        min[i++] = c;  
    min[i++] = '\0';  
  
    ++i;  
  
    /* find the last value in the range */  
    while(isalnum(c = word[i++]))  
        max[j++] = c;  
    max[j] = '\0';  
  
    /* evaluate the first value in the range */  
    switch(evaluate(min))  
    {  
    case SIMPENUM: temprange.first = eval_simple_num(min); break;  
    case NATURALNUM: temprange.first = eval_natural_num(min); break;  
    case COMPLEXNUM: temprange.first = eval_complex_num(min); break;  
    case UARNAME: temprange.first = LOST; break;  
    case UNKNOWN: temprange.first = LOST; break;  
    default: temprange.first = LOST; break;  
    }  
  
    /* evaluate the last value in the range */  
    switch(evaluate(max))  
    {  
    case SIMPENUM: temprange.last = eval_simple_num(max); break;  
    case NATURALNUM: temprange.last = eval_natural_num(max); break;  
    case COMPLEXNUM: temprange.last = eval_complex_num(max); break;  
    case UARNAME: temprange.last = LOST; break;  
    }
```

Appendix D

```
case UNKNOWN: temprange.last = LOST; break;
default: temprange.last = LOST; break;
}

/* calculate the range */
if((temprange.first != LOST) && (temprange.last != LOST))
    temprange.span = temprange.last - temprange.first + 1;
else
    temprange.span = LOST;

return(temprange.span);
}

/*-----
CALLNAME: EVAL_NATURAL_NUM
AUTHOR: S. Treadwell
CREATED: 4 MAY 92
UPDATED: 15 JUN 92

Will take an expression like 'natural(number)' and evaluate its value based
upon the number within the parentheses. This value is then returned as the
value of the expression.
-----*/

int eval_natural_num(word)
char word[40];
{
    int evaluate();
    int eval_simple_num();
    int eval_complex_num();
    int c;
    int i = 0;
    int j = 0;
    int value = LOST;
    char argument[40];

    if (word[7] == '(')
    {
        i = 8;
        while((c = word[i++]) != ')')
            argument[j++] = c;
        argument[j] = '\0';
        switch(evaluate(argument))
        {
            case SIMPLENUM: value = eval_simple_num(argument); break;
            case COMPLEXNUM: value = eval_complex_num(argument); break;
            case VARNAME: value = LOST; break;
            case UNKNOWN: value = LOST; break;
            default: value = LOST; break;
        }
    }
    return(value);
}
```

Appendix D

```
/*-----  
CALLNAME: EVAL_COMPLEX_NUM  
AUTHOR: S. Treadwell  
CREATED: 4 MAY 92  
UPDATED: 15 JUN 92  
  
Will take a string expression like '16*ffff#' and parse it into a single  
integer value and return that value as the output of the function.  
The string must be in a form almost exactly like that given. Any base or  
internal value can be used, provided that it is not too large to be held  
in an integer.  
-----*/
```

```
int eval_complex_num(word)  
char word[];  
{  
    int i = 0;  
    int j = 0;  
    int c;  
    int basevalue;  
    int numvalue = LOST;  
    char base[20];  
    char num[20];  
  
    while(isdigit(c = word[i]))  
        base[i++] = c;  
    base[i++] = '\0';  
  
    while(isxdigit(c = word[i++]))  
        num[j++] = c;  
    num[j] = '\0';  
  
    basevalue = strtol(base,(char **)NULL,10);  
    numvalue = strtol(num,(char **)NULL,basevalue);  
  
    return(numvalue);  
}
```

```
/*-----  
CALLNAME: EVAL_SIMPLE_NUM  
AUTHOR: S. Treadwell  
CREATED: 4 MAY 92  
UPDATED: 5 MAY 92  
  
Will take a pure number in string form and convert it to integer form and  
return that integer as the output of the function.  
Cannot deal with floating point numbers; only integers are allowed.  
Assumes that everything expressed as a pure number is in base 10.  
-----*/
```

```
int eval_simple_num(word)  
char word[];  
{  
    int value = LOST;  
  
    value = strtol(word,(char **)NULL,10);  
    return(value);  
}
```

Appendix D

```
/*-----
CALLNAME: EVALUATE
AUTHOR: S. Treadwell
CREATED: 4 MAY 92
UPDATED: 5 MAY 92

Will take a string as an input argument, evaluate the contents of that string
and put it in a class according to its configuration. The class is returned
as an integer value and the classes are defined in "header.h"
Works for all tested cases, but may not be foolproof. A safety net is
provided through the UNKNOWN class of string, which is triggered when the
string does not fit any of the allowed patterns.
-----*/

int evaluate(word)
char word[];
{
    int class = UNKNOWN;
    int i = 0;
    int k = 0;
    int c;
    int dots = 0;
    int pound = 0;
    int other = 0;

    if (isdigit(word[0]))
    {
        while((c = word[i++]) != '\0')
        {
            if(c == '.') ++dots;
            if(c == '#') ++pound;
            if(!isdigit(c)) ++other;
        }
        if ((dots == 2) & (pound == 0) & (other == 2)) class = RANGENUM;
        if ((dots == 0) & (pound == 2) & (other >= 2)) class = COMPLEXNUM;
        if ((dots == 0) & (pound == 0) & (other == 0)) class = SIMPLENUM;
    }

    if (isalpha(word[0]))
    {
        class = UARNAME;
        if((k = strncmp(word,"natural",7)) == 0) class = NATURALNUM;
        while((c = word[i++]) != '\0')
            if(c == '.')
                if((c = word[i++]) == '.') class = RANGENUM;
    }
    return(class);
}
```


Appendix D

```
/*-----
CALLNAME: PRINT_PROCEDURES
AUTHOR: S. Treadwell
CREATED: 19 OCT 92
UPDATED: 19 OCT 92

Will print out the models for each of the procedures found for a given task
This procedure is provided for filling the error log with valuable info for
tracking down analysis and/or run-time errors.
-----*/

void print_procedures(procedures,error_file)
struct proc_list_info *procedures;
FILE *error_file;
{
    int i,j,k;

    for(i = 0; i < procedures->length; ++i)
    {
        fprintf(error_file,"Procedure %s found in %s\n",procedures->entry[i].name,
            procedures->entry[i].filename);
        for(j = 0; j < procedures->entry[i].skeleton.length; ++j)
            fprintf(error_file,"Type:%3d Value:%4d Depth:%3d Pointer:%3d\n",
                procedures->entry[i].skeleton.entry[j].type,
                procedures->entry[i].skeleton.entry[j].value,
                procedures->entry[i].skeleton.entry[j].depth,
                procedures->entry[i].skeleton.entry[j].pointer);
        for(j = 0; j < procedures->entry[i].skeleton.num_counters; ++j)
            fprintf(error_file,"Ctr %2d: Queued %3d Rtrud %3d Sent %3d Read %3d\n",j,
                procedures->entry[i].skeleton.counter_set[j].num_queued,
                procedures->entry[i].skeleton.counter_set[j].num_retrieved,
                procedures->entry[i].skeleton.counter_set[j].num_sent,
                procedures->entry[i].skeleton.counter_set[j].num_read);
    }
}

void clear_screen()
{
    printf("\033[%dJ",2);
    printf("\n");
}
```

Appendix D

```

/*-----
CALLNAME: FIND_WORST_PATH
AUTHOR: S. Treadwell
CREATED: 01 SEP 92
UPDATED: 20 OCT 92

For a given model, this procedure controls model preparation, reduction, and
path generation. The execution path generation leads to identification of
the worst case path and the parameterization of that path is returned to the
parent procedure, process_list.
-----*/

struct counter_list find_worst_path(skeleton,delay_data,error_file)
struct model_info *skeleton;
struct constant_list *delay_data;
FILE *error_file;
{
    void reduce_model();
    int calculate_time();
    enum boolean model_ok();
    struct counter_list generate_paths();
    struct counter_list big_counter, new_counter;
    int i;

    /* Initialize path parameterization */
    big_counter.num_queued = 0;
    big_counter.num_retrieved = 0;
    big_counter.num_sent = 0;
    big_counter.num_read = 0;
    big_counter.total_time = 0;
    big_counter.num_msg_queued = 0;

    /* prepare and reduce the model */
    reduce_model(skeleton,delay_data,error_file);

    /* include the final task model in the error log */
    fprintf(error_file,"Task Model...\n");
    for(i = 0; i < skeleton->length; ++i)
        fprintf(error_file,"%3d Type:%3d Uvalue:%4d Depth:%3d Pointer:%3d\n",i,
            skeleton->entry[i].type,skeleton->entry[i].value,
            skeleton->entry[i].depth,skeleton->entry[i].pointer);
    for(i = 0; i < skeleton->num_counters; ++i)
        fprintf(error_file,"Ctr %2d: Queued %3d Retvd %3d Sent %3d Read %3d\n",
            i,skeleton->counter_set[i].num_queued,
            skeleton->counter_set[i].num_retrieved,
            skeleton->counter_set[i].num_sent,
            skeleton->counter_set[i].num_read);

    if(model_ok(skeleton,error_file)) /* trap for model errors */
    {
        /* generate all possible paths that start at the top of the model */
        big_counter = generate_paths(skeleton,delay_data,0,2,error_file);

        /* generate paths beginning with each WFS entry in the model */
        for(i = 0; i < skeleton->length; ++i)
            if(skeleton->entry[i].type == WFS)
            {
                new_counter = generate_paths(skeleton,delay_data,i+1,2,error_file);
            }
    }
}

```

Appendix D

```
        if (calculate_time(delay_data,&new_counter) >
            calculate_time(delay_data,&big_counter))
            big_counter = new_counter;
    }
}
else fprintf(error_file,"The model could not be processed due to faults\n");

return(big_counter);
}

/*-----
CALLNAME: REDUCE_MODEL
AUTHOR: S. Treadwell
CREATED: 06 OCT 92
UPDATED: 19 OCT 92

This procedure takes a given model and completes it's development. It then
reduces it by squeezing out loops, if's, and case statements that have no
WFS inside. These constructs are replaced by counter_set entries.
-----*/

void reduce_model(skeleton,delay_data,error_file)
struct model_info *skeleton;
struct constant_list *delay_data;
FILE *error_file;
{
    void crunch();
    void check_ctrs();
    void nest_level();
    void match_loops();
    int i;

    /* model preparation */
    nest_level(skeleton);
    match_loops(skeleton);

    /* for(i = 0; i < skeleton->length; ++i)
        fprintf(error_file,"%3d Type:%3d Uvalue:%4d Depth:%3d Pointer:%3d\n",i,
            skeleton->entry[i].type,skeleton->entry[i].value,
            skeleton->entry[i].depth,skeleton->entry[i].pointer);**/

    /* model reduction */
    crunch(skeleton,delay_data,LOOP,error_file);
    crunch(skeleton,delay_data,WHILE_LOOP,error_file);
    crunch(skeleton,delay_data,FOR_LOOP,error_file);
    crunch(skeleton,delay_data,IF,error_file);
    crunch(skeleton,delay_data,CASE,error_file);

    /* eliminate any empty counter_set entries */
    check_ctrs(skeleton,error_file);
}
}
```

Appendix D

```
/*-----  
CALLNAME: NEST_LEVEL  
AUTHOR: S. Treadwell  
CREATED: 07 AUG 92  
UPDATED: 04 MAR 93  
  
Takes the abstracted model and determines the level of nesting for each item  
in the model. This is necessary for the process of matching loops and other  
control flow items. Each type of model entry has a particular effect on the  
nesting value for entries that follow.  
-----*/
```

```
void nest_level(skeleton)  
struct model_info *skeleton;  
{  
    int i;  
    int nest = 0;  
  
    for (i = 0; i < skeleton->length; ++i)  
        switch(skeleton->entry[i].type)  
        {  
            case LOOP:  
                skeleton->entry[i].depth = nest;  
                ++nest;  
                break;  
            case FOR_LOOP:  
                skeleton->entry[i].depth = nest;  
                ++nest;  
                break;  
            case WHILE_LOOP:  
                skeleton->entry[i].depth = nest;  
                ++nest;  
                break;  
            case IF:  
                skeleton->entry[i].depth = nest;  
                ++nest;  
                break;  
            case ELSIF:  
                --nest;  
                skeleton->entry[i].depth = nest;  
                ++nest;  
                break;  
            case CASE:  
                skeleton->entry[i].depth = nest;  
                ++nest;  
                break;  
            case WHEN:  
                --nest;  
                skeleton->entry[i].depth = nest;  
                ++nest;  
                break;  
            case ELSE:  
                --nest;  
                skeleton->entry[i].depth = nest;  
                ++nest;  
                break;  
            case END_LOOP:  
                --nest;
```

Appendix D

```
    skeleton->entry[i].depth = nest;
    break;
case END_IF:
    --nest;
    skeleton->entry[i].depth = nest;
    break;
case END_CASE:
    --nest;
    skeleton->entry[i].depth = nest;
    break;
default:
    skeleton->entry[i].depth = nest;
}
}
```

Appendix D

```
/*-----  
CALLNAME: MATCH_LOOPS  
AUTHOR: S. Treadwell  
CREATED: 07 AUG 92  
UPDATED: 04 MAR 93  
  
This procedure will look through the model and match loop statements with  
the proper end_loop statements. Likewise for the if end_if and case  
end_case pairs. Other matchings are necessary as well. Note that empty  
loops are eliminated in the beginning. It may be necessary to do the same  
with empty if's and empty case statements. The matching is accomplished  
through the establishment of the pointer elements in model entries.  
-----*/  
  
void match_loops(skeleton)  
struct model_info *skeleton;  
{  
    int loop_starts[20];  
    int num_loops = 0;  
    int i,j;  
  
    for (i = 0; i < skeleton->length; ++i)  
        skeleton->entry[i].pointer = UNDEFINED;  
  
    /* Eliminate empty loops */  
    for (i = 0; i < skeleton->length; ++i)  
        switch(skeleton->entry[i].type)  
        {  
            case FOR_LOOP:  
                if(skeleton->entry[i+1].type == END_LOOP)  
                {  
                    for(j = i; j < skeleton->length - 2; ++j)  
                        skeleton->entry[j] = skeleton->entry[j+2];  
                    skeleton->length -= 2;  
                }  
                break;  
            case WHILE_LOOP:  
                if(skeleton->entry[i+1].type == END_LOOP)  
                {  
                    for(j = i; j < skeleton->length - 2; ++j)  
                        skeleton->entry[j] = skeleton->entry[j+2];  
                    skeleton->length -= 2;  
                }  
                break;  
            default: break;  
        }  
  
    /* loop_starts is an array of index values for entries that represent */  
    /* the beginning of a loop construct. It simplifies the matching of */  
    /* loop starts and end_loop entries. */  
    for (i = 0; i < skeleton->length; ++i)  
        switch(skeleton->entry[i].type)  
        {  
            case LOOP: loop_starts[num_loops++] = i; break;  
            case FOR_LOOP: loop_starts[num_loops++] = i; break;  
            case WHILE_LOOP: loop_starts[num_loops++] = i; break;  
            case END_LOOP:  
                skeleton->entry[i].pointer = loop_starts[--num_loops];  
        }  
}
```

Appendix D

```

    skeleton->entry[skeleton->entry[i].pointer].pointer = i;
    break;
case END_IF: /* match with the corresponding IF statement */
    for(j = i-1; j >= 0; --j)
        if((skeleton->entry[j].type == IF)&&
            (skeleton->entry[j].depth == skeleton->entry[i].depth))
            {
                skeleton->entry[j].pointer = i;
                skeleton->entry[i].pointer = j;
                break;
            }
    break;
case END_CASE: /* match with the corresponding CASE entry */
    for(j = i-1; j >= 0; --j)
        if((skeleton->entry[j].type == CASE)&&
            (skeleton->entry[j].depth == skeleton->entry[i].depth))
            {
                skeleton->entry[j].pointer = i;
                skeleton->entry[i].pointer = j;
                break;
            }
    break;
case WHEN: /* point to next WHEN or END_CASE entry */
    for(j = i+1; j < skeleton->length; ++j)
        if(((skeleton->entry[j].type == WHEN)||
            (skeleton->entry[j].type == END_CASE)) &&
            (skeleton->entry[j].depth == skeleton->entry[i].depth))
            {
                skeleton->entry[i].pointer = j;
                break;
            }
    break;
case IF: /* point to next branch in the "if" construct */
    for(j = i+1; j < skeleton->length; ++j)
        if(((skeleton->entry[j].type == ELSE)||
            (skeleton->entry[j].type == ELSIF)||
            (skeleton->entry[j].type == END_IF)) &&
            (skeleton->entry[j].depth == skeleton->entry[i].depth))
            {
                skeleton->entry[i].pointer = j;
                break;
            }
    break;
case ELSIF: /* point to next branch in the "if" construct */
    for(j = i+1; j < skeleton->length; ++j)
        if(((skeleton->entry[j].type == ELSE)||
            (skeleton->entry[j].type == ELSIF)||
            (skeleton->entry[j].type == END_IF)) &&
            (skeleton->entry[j].depth == skeleton->entry[i].depth))
            {
                skeleton->entry[i].pointer = j;
                break;
            }
    break;
case ELSE: /* point to the end of the "if" construct */
    for(j = i+1; j < skeleton->length; ++j)
        if((skeleton->entry[j].type == END_IF) &&
            (skeleton->entry[j].depth == skeleton->entry[i].depth))
            {

```

Appendix D

```
        skeleton->entry[i].pointer = j;
        break;
    }
    break;

    default: break;
}
}

/*-----
CALLNAME: CRUNCH
AUTHOR: S. Treadwell
CREATED: 01 SEP 92
UPDATED: 19 OCT 92

This procedure identifies constructs that do not contain WFS calls and
crunches them into a set of counters to replace the construct. This is
necessary simplification because it allows loops that contain critical
constructs to be counted for their total iterations, not just a single
pass through.
-----*/

void crunch(skeleton,delay_data,start_type,error_file)
int start_type;
struct model_info *skeleton;
struct constant_list *delay_data;
FILE *error_file;
{
    struct counter_list generate_paths();
    struct model_info temp;
    int i,j;
    int k = 0;
    int begin;
    int end;
    enum boolean present = NO;
    enum boolean qualified = NO;

    for(i = 0; i < skeleton->length; ++i)
    {
        /* Guard against crunching loops with undefined iteration maximums */
        qualified = NO;
        if(skeleton->entry[i].type == start_type)
            switch(start_type)
            {
                case IF: qualified = YES; break;
                case CASE: qualified = YES; break;
                default: if(skeleton->entry[i].value > 0) qualified = YES; break;
            }

        /* Find the bounds on the targeted construct */
        if (qualified == YES)
        {
            begin = i;
            for(j = i+1; j < skeleton->length; ++j)
            {
                if(skeleton->entry[j].type == WFS) present = YES;
                if(skeleton->entry[j].pointer == begin)
                {
```


Appendix D

```
/*-----  
CALLNAME: CHECK_CTRS  
AUTHOR: S. Treadwell  
CREATED: 16 NOV 92  
UPDATED: 16 NOV 92  
  
Will examine the skeleton sent in and check for COUNTER_SET entries  
containing only zeros. These will be removed from the model.  
-----*/
```

```
void check_ctrs(skeleton,error_file)  
struct model_info *skeleton;  
FILE *error_file;  
{  
    int i,j,k;  
    int sum;  
  
    for(i = skeleton->num_counters - 1; i >= 0; --i)  
    {  
        sum = 0;  
        sum = skeleton->counter_set[i].num_queued +  
            skeleton->counter_set[i].num_retrieved +  
            skeleton->counter_set[i].num_sent +  
            skeleton->counter_set[i].num_read;  
        if(sum == 0) /* eliminate the entry and adjust the model accordingly */  
        {  
            --skeleton->num_counters;  
            for(j = i; j < skeleton->num_counters; ++j)  
                skeleton->counter_set[j] = skeleton->counter_set[j+1];  
            for(j = 0; j < skeleton->length; ++j)  
                if((skeleton->entry[j].type == COUNTERSET)&&  
                    (skeleton->entry[j].value == i))  
                {  
                    --skeleton->length;  
                    for(k = j; k < skeleton->length; ++k)  
                        skeleton->entry[k] = skeleton->entry[k+1];  
                }  
            for(j = 0; j < skeleton->length; ++j)  
                if((skeleton->entry[j].type == COUNTERSET)&&  
                    (skeleton->entry[j].value > i))  
                    skeleton->entry[j].value -= 1;  
        }  
    }  
}
```

Appendix D

```
/*-----  
CALLNAME: GENERATE_PATHS  
AUTHOR: S. Treadwell  
CREATED: 15 AUG 92  
UPDATED: 19 OCT 92  
  
This procedure generates all possible execution paths through a given model.  
It begins at the designated starting entry and exhausts all paths from that  
point using the decision integer as the path determinator. Once a path is  
defined, it is parameterized, quantified, and compared to previous paths.  
-----*/
```

```
struct counter_list generate_paths(skeleton, delay_data, start, mode, error_file)  
struct model_info *skeleton;  
struct constant_list *delay_data;  
int start;  
int mode;  
FILE *error_file;  
{  
    int calculate_time();  
    int decide();  
    struct counter_list parameterize();  
    struct counter_list big_counter, new_counter;  
    int path[30];  
    int j = 0;  
    int i, k;  
    int done;  
    int next;  
    int shift_num;  
    unsigned long temp;  
    unsigned long decision = 0x00000000;  
  
    big_counter.num_queued = 0;  
    big_counter.num_retrieved = 0;  
    big_counter.num_sent = 0;  
    big_counter.num_read = 0;  
    big_counter.num_msg_queued = 0;  
    big_counter.total_time = 0;  
  
    while(decision < 0x80000000) /*decision=80000000 means all paths exhausted*/  
    {  
        for(i = 0; i < skeleton->length; ++i)  
            skeleton->entry[i].flow = BLANK;  
        i = start; shift_num = 30; done = NO;  
        do  
        {  
            if ((mode == 1) && (i == skeleton->length - 1))        done = YES;  
            switch(skeleton->entry[i].flow)  
            {  
                case EXEC: next = i + 1; path[j++] = i; break;  
                case NO_EXEC: next = i + 1; break;  
                default:  
                    path[j] = i; /* printf("i is %d\n", i); **/  
                    switch(skeleton->entry[i].type)  
                    {  
                        case WHILE_LOOP:  
                            next = decide(skeleton, i, &j, decision, &shift_num);  
                            break;  
                    }  
            }  
        }  
    }  
}
```

Appendix D

```

case END_LOOP:
    if(mode == 1) next = i + 1;
    else
    {
        if(skeleton->entry[skeleton->entry[i].pointer].value == INFINITE)
        {
            next = skeleton->entry[i].pointer;
            ++j;
        }
        else next = decide(skeleton,i,&j,decision,&shift_num);
    }
    break;
case IF:
    next = decide(skeleton,i,&j,decision,&shift_num);
    break;
case ELSE:
    next = i + 1;
    ++j;
    break;
case ELSIF:
    next = decide(skeleton,i,&j,decision,&shift_num);
    break;
case WHEN:
    next = decide(skeleton,i,&j,decision,&shift_num);
    break;
case WFS:
    ++j;
    done = YES;
    break;
default:
    ++j;
    next = i + 1;
    break;
}
break;
}
i = next;
}
while(!done);

if(mode == 2) /* mode 2 is for full task model analysis */
{
    fprintf(error_file,"PATH:");
    for(i = 0; i < j; ++i)
        fprintf(error_file,"%3d",path[i]);
    fprintf(error_file,"\n");
}

/* parameterize the path just completed and compare to previous paths */
new_counter = parameterize(skeleton,path,j,mode);
if(calculate_time(delay_data,&new_counter) >
    calculate_time(delay_data,&big_counter))
    big_counter = new_counter;
j = 0;
++shift_num; /* a necessary adjustment */
/* now update the decision integer to determine the next path */
/* the math here effectively reverses the last decision made */
/* in the last execution path */
decision &= (0xffffffff << shift_num);

```

Appendix D

```
    decision += (0x01 << shift_num);
}

/* big_counter is the parameterization of the worst case path through */
/* the given model for the given starting point */
return(big_counter);
}

/*-----*/
CALLNAME: DECIDE
AUTHOR: S. Treadwell
CREATED: 16 AUG 92
UPDATED: 15 SEP 92

This procedure makes the decision of where to go next according to the
decision integer. The next point in the path is returned as an integer.
The shift number is updated if a decision is made but the decision
integer is not changed until the path is complete.
-----*/

int decide(skeleton,i,j,decision,shift_num)
struct model_info *skeleton;
int i;
int *j;
long int decision;
int *shift_num;
{
    int next,k,l;
    int iterations;
    int first_no_exec;
    unsigned choice;

    choice = (decision >> *shift_num) & 0x01;
    if(choice) choice = EXEC;
    else choice = NO_EXEC;
    *shift_num -= 1;

    switch(choice)
    {
    case EXEC:
        next = i + 1;
        ++*j;
        /* must block off the unchosen branch of the construct */
        k = skeleton->entry[i].pointer;
        first_no_exec = k;
        switch(skeleton->entry[i].type)
        {
        case IF:
            while(skeleton->entry[k].type != END_IF)
                k = skeleton->entry[k].pointer;
            for(l = first_no_exec; l < k; ++l)
                skeleton->entry[l].flow = NO_EXEC;
            break;
        case ELSIF:
            while(skeleton->entry[k].type != END_IF)
                k = skeleton->entry[k].pointer;
            for(l = first_no_exec; l < k; ++l)
                skeleton->entry[l].flow = NO_EXEC;
        }
    }
}
```

Appendix D

```
    break;
case WHEN:
    while(skeleton->entry[k].type != END_CASE)
        k = skeleton->entry[k].pointer;
    for(l = first_no_exec; l < k; ++l)
        skeleton->entry[l].flow = NO_EXEC;
    break;
default: break;
}
break;
case NO_EXEC:
    /* jump to the next branch using the pointer element */
    next = skeleton->entry[i].pointer;
    switch(skeleton->entry[i].type)
    {
    case WHILE_LOOP:
        ++next;
        break;
    case END_LOOP:
        skeleton->entry[i].flow++;
        ++*j;
        /* need to guard against troublesome constructs and ensure that a */
        /* loop containing a conditional WFS entry does not become an */
        /* infinite loop when generating paths */
        iterations = skeleton->entry[skeleton->entry[i].pointer].value;
        if(iterations > 0)
            if(skeleton->entry[i].flow == iterations - 1)
                skeleton->entry[i].flow = EXEC;
            if(skeleton->entry[i].flow == 5) /* 5 is a stringent limit */
                skeleton->entry[i].flow = EXEC;
        break;
    default: break;
    }
    break;
}
return(next);
}
```

Appendix D

```
/*-----  
CALLNAME: PARAMETERIZE  
AUTHOR: S. Treadwell  
CREATED: 7 SEP 92  
UPDATED: 9 SEP 92  
  
Will take a given path through the model and add up all the critical system  
calls along the path. This info is stored as a set of counters and  
returned to the parent procedure.  
-----*/  
  
struct counter_list parameterize(skeleton,path,path_length,mode)  
struct model_info *skeleton;  
int path[];  
int path_length;  
int mode;  
{  
    int i,k;  
    int j = 0;  
    int loop[10];  
    int iterations;  
    int counter_num;  
    struct counter_list counter_set;  
  
    counter_set.num_queued = 0;  
    counter_set.num_retrieved = 0;  
    counter_set.num_sent = 0;  
    counter_set.num_read = 0;  
    counter_set.num_msg_queued = 0;  
    counter_set.msg_retrieved = 0;  
  
    for(i = 0; i < path_length; ++i)  
    {  
        iterations = 1;  
  
        if(mode == 1)  
            for(k = 0; k < j; ++k)  
                iterations *= loop[k];  
  
        switch(skeleton->entry[path[i]].type)  
        {  
            case LOOP: loop[j++] = skeleton->entry[path[i]].value; break;  
            case FOR_LOOP: loop[j++] = skeleton->entry[path[i]].value; break;  
            case WHILE_LOOP: loop[j++] = skeleton->entry[path[i]].value; break;  
            case END_LOOP: if(j > 0) j--; break; /* for mode 2--will explain later */  
            case COUNTERSET:  
                counter_num = skeleton->entry[path[i]].value;  
                counter_set.num_queued += skeleton->counter_set[counter_num].num_queued;  
                counter_set.num_retrieved +=  
                    skeleton->counter_set[counter_num].num_retrieved;  
                counter_set.num_sent += skeleton->counter_set[counter_num].num_sent;  
                counter_set.num_read += skeleton->counter_set[counter_num].num_read;  
                counter_set.num_msg_queued +=  
                    skeleton->counter_set[counter_num].num_msg_queued;  
                counter_set.msg_retrieved +=  
                    skeleton->counter_set[counter_num].msg_retrieved;  
                break;  
            case QUEUE:
```

Appendix D

```
if (iterations > 0)
{
    counter_set.num_queued +=(skeleton->entry[path[i]].value * iterations);
    counter_set.num_msg_queued += iterations;
}
else
{
    counter_set.num_queued += (skeleton->entry[path[i]].value * 10);
    counter_set.num_msg_queued += 10;
}
break;
case RETRIEVE:
if(iterations > 0)
{
    counter_set.num_retrieved+=(skeleton->entry[path[i]].value*iterations);
    counter_set.msg_retrieved+=iterations;
}
else
{
    counter_set.num_retrieved += (skeleton->entry[path[i]].value * 10);
    counter_set.msg_retrieved+=10;
}
break;
case SEND:
if (iterations > 0)
    counter_set.num_sent += (skeleton->entry[path[i]].value * iterations);
else
    counter_set.num_sent += (skeleton->entry[path[i]].value * 10);
break;
case READ:
if (iterations > 0)
    counter_set.num_read+=(skeleton->entry[path[i]].value*iterations);
else
    counter_set.num_read += (skeleton->entry[path[i]].value * 10);
break;
default: break;
}
}
return(counter_set);
}
```


Appendix D

```
/*-----  
CALLNAME: MODEL_OK  
AUTHOR: S. Treadwell  
CREATED: 20 OCT 92  
UPDATED: 12 NOV 92  
  
Will verify the structure of the model to ensure that it is prepared for  
analysis.  
-----*/  
  
enum boolean model_ok(skeleton,error_file)  
struct model_info *skeleton;  
FILE *error_file;  
{  
    int i,j;  
    int loops = 0;  
    int ifs = 0;  
    int cases = 0;  
    int waits = 0;  
    enum boolean valid = YES;  
  
    /* Make sure the model begins and ends with depths = 0 */  
    if(skeleton->entry[skeleton->length - 1].depth != 0)  
    {  
        valid = NO;  
        fprintf(error_file,"The final model entry is at the wrong depth\n");  
    }  
    if(skeleton->entry[0].depth != 0)  
    {  
        valid = NO;  
        fprintf(error_file,"The initial model entry is at the wrong depth\n");  
    }  
  
    /* Ensure that for every loop there is an end_loop and the same with */  
    /* case statements and if constructs */  
    for(i = 0; i < skeleton->length; ++i)  
        switch(skeleton->entry[i].type)  
        {  
            case LOOP: ++loops; break;  
            case FOR_LOOP: ++loops; break;  
            case WHILE_LOOP: ++loops; break;  
            case END_LOOP: --loops; break;  
            case IF: ++ifs; break;  
            case END_IF: --ifs; break;  
            case CASE: ++cases; break;  
            case END_CASE: --cases; break;  
            case WFS: ++waits; break;  
            case COUNTERSET:  
                /* make sure the model does not reference a non-existent counter set */  
                if(skeleton->entry[i].value >= skeleton->num_counters)  
                {  
                    valid = NO;  
                    fprintf(error_file,  
                        "Excessive counter_set number at model entry %d\n",i);  
                }  
                break;  
            default: break;  
        }  
}
```

Appendix D

```
if(loops != 0)
{
    valid = NO;
    fprintf(error_file,"There is improper loop matching in the model\n");
}
if(cases != 0)
{
    valid = NO;
    fprintf(error_file,"There is improper case matching in the model\n");
}
if(ifs != 0)
{
    valid = NO;
    fprintf(error_file,"There is improper if/end_if matching in the model\n");
}
if(waits == 0) /* ensure that the task model includes a WFS call */
{
    valid = NO;
    fprintf(error_file,"There is no WFS in this model\n");
}

/* Make sure there are no infinite loops containing no WFS calls */
for(i = 0; i < skeleton->length; ++i)
    if((skeleton->entry[i].type == LOOP)&&(skeleton->entry[i].value<0))
    {
        waits = 0;
        for(j = i+1; j < skeleton->length; ++j)
        {
            if(skeleton->entry[j].type == WFS) ++waits;
            if((skeleton->entry[j].type == END_LOOP)&&
                (skeleton->entry[j].pointer == i))
                break;
        }
        if (waits == 0)
        {
            valid = NO;
            fprintf(error_file,"Infinite loop containing no WFS\n");
        }
    }

/* If the model passes all tests, valid = YES; otherwise, valid = NO */
return(valid);
}
```

Appendix D

```
/*-----  
CALLNAME: CALCULATE_TIME  
AUTHOR: S. Treadwell  
CREATED: 09 SEP 92  
UPDATED: 30 OCT 92  
  
This procedure takes a path parameterization and calculates a lower bound  
on delay for that path using system specific delay data.  
-----*/  
  
int calculate_time(delay_data,counter_set)  
struct constant_list *delay_data;  
struct counter_list *counter_set;  
{  
    int sum = 0;  
    int extra = 0;  
  
    if(counter_set->num_queued>0) sum +=  
        (delay_data->queue_coeff*counter_set->num_queued)+  
        (delay_data->queue_const*counter_set->num_msg_queued);  
  
    if(counter_set->num_sent>0) sum +=  
        delay_data->queue_coeff*counter_set->num_sent+delay_data->queue_const;  
  
    if(counter_set->num_retrieved>0) sum +=  
        (delay_data->retrieve_coeff*counter_set->num_retrieved)+  
        (delay_data->retrieve_const*counter_set->msg_retrieved);  
  
    if(counter_set->num_read>0) sum +=  
        delay_data->retrieve_coeff*counter_set->num_read+  
        delay_data->retrieve_const;  
  
    counter_set->total_time = sum;  
  
    /* Certain critical constructs add delay to the frame overhead as well as */  
    /* incurring task execution delay. It is critical to account for this */  
    /* additional delay when comparing various paths. This extra delay is */  
    /* not added into the "total_time" parameter but it is part of the "sum" */  
    /* that is returned to the parent procedure */  
  
    extra += delay_data->IH_coeff * counter_set->num_queued;  
    if(counter_set->num_queued > 0)  
        extra += delay_data->RGD_msg_coeff * counter_set->num_queued +  
        delay_data->RGD_msg_coeff * counter_set->num_msg_queued;  
    else extra += 21;  
  
    sum += extra;  
  
    return(sum);  
}
```

Appendix E

External Files

key_words.dat

```
scheduler.wait_for_schedule
loop
if
else
elsif
case
when
rg_communication.queue_message
rg_communication.retrieve_message
rg_communication.send_message
rg_communication.read_message
task
:=
range
package
procedure
begin
end
gtid
vg
rg
max_xmit_size
max_xmit_num
max_rcve_size
max_rcve_num
select
record
accept
```

constants.dat

```
queue_coefficient 45
queue_constant 43
retrieve_coefficient 61
retrieve_constant 67
interrupt_handler_coefficient 110
interrupt_handler_constant 103
RGD_message_coefficient 123
RGD_message_constant -12
RGD_task_coefficient 26
RGD_task_constant 15
RGD_overall_constant 49
RGD_empty_queue_constant 70
context_switch 19
local_FDIR 84
system_FDIR 1316
```


Appendix F

An Illustrative Example

1. The files included in this section are taken directly from a full-scale test of the AFTA timing analysis tool.
2. The task specification file is `task_list.adb`.
3. The source code for the application tasks is found in the following files:
 - `app_test.adb`
 - `sys_fdi.adb`
 - `test_code.adb`
4. Note that the task called `test_t` is not intended to represent actual application task code. It is used merely for testing purposes and is designed to highlight some of the features of the timing analysis tool that are not fully exercised by the legitimate tasks: `appl1_t`, `appl2_t`, and `sys_fdi_t`.
5. The following files are intermediate files passed from the preliminary processing stage to the software and hardware analysis stages:
 - `task_names.dat`
 - `list_of_tasks.dat`
 - `filenames.dat`
6. The output of the analysis is found in `errors.dat` and `results.dat`.

task_list.ada

```

with config;
with ne_interface; use ne_interface;
with gtids;
with gcids;

-----
--
-- System task specification.
--
-----
package task_list is

-- specification of rg tasks in system; does not include rg and io dispatchers;
-- they are special rg4 tasks and are specified within config.init_cid_config
--
-- NOTE:
-- There is no ordering requirement in list, but an ordering convention makes
-- the list easier to read. The implemented convention is to order the task
-- based on vg,rg, and precedence.
task_list : constant config.task_list_r := (
  num_tasks => 10,
  tasks => (

-- tasks only on vg 0
  1 => (
    gcid => gcids.appl1_1,
    gtid => gtids.appl1,
    location => config.one_vg,
    vg => 0,
    rg => config.rg4,
    precedence => 0,
    max_xmit_size => 400,
    max_xmit_num => 5,
    max_rcve_size => 400,
    max_rcve_num => 20,
    num_iors => 0,
    iors => (
      others => (
        num_chains => 0,
        chains => (
          others => (E=>false,D=>false,C=>false,B=>false,A=>false))))));

  2 => (
    gcid => gcids.appl2_1,
    gtid => gtids.appl2,
    location => config.one_vg,
    vg => 0,
    rg => config.rg4,
    precedence => 4,
    max_xmit_size => 200,
    max_xmit_num => 10,
    max_rcve_size => 200,
    max_rcve_num => 20,
    num_iors => 0,
    iors => (
      others => (
        num_chains => 0,

```

Appendix F

```
        chains => (
            others => (E=>false,D=>false,C=>false,B=>false,A=>false))))),
3 => (
    gcid => gcids.system_fdi,
    gtid => gtids.system_fdi,
    location => config.one_vg,
    vg => 0,
    rg => config.rg4,
    precedence => 14,
    max_xmit_size => 200,
    max_xmit_num => 10,
    max_rcve_size => 200,
    max_rcve_num => 20,
    num_iors => 0,
    iors => (
        others => (
            num_chains => 0,
            chains => (
                others => (E=>false,D=>false,C=>false,B=>false,A=>false))))),
4 => (
    gcid => gcids.appl1_2,
    gtid => gtids.appl1,
    location => config.one_vg,
    vg => 0,
    rg => config.rg3,
    precedence => 3,
    max_xmit_size => 200,
    max_xmit_num => 10,
    max_rcve_size => 200,
    max_rcve_num => 20,
    num_iors => 0,
    iors => (
        others => (
            num_chains => 0,
            chains => (
                others => (E=>false,D=>false,C=>false,B=>false,A=>false))))),
5 => (
    gcid => gcids.appl2_2,
    gtid => gtids.appl2,
    location => config.one_vg,
    vg => 0,
    rg => config.rg3,
    precedence => 4,
    max_xmit_size => 200,
    max_xmit_num => 10,
    max_rcve_size => 200,
    max_rcve_num => 20,
    num_iors => 0,
    iors => (
        others => (
            num_chains => 0,
            chains => (
                others => (E=>false,D=>false,C=>false,B=>false,A=>false))))),
6 => (
    gcid => gcids.appl1_3,
    gtid => gtids.appl1,
    location => config.one_vg,
    vg => 0,
    rg => config.rg2,
```


Appendix F

```
precedence => 0,
max_xmit_size => 400,
max_xmit_num => 5,
max_rcve_size => 200,
max_rcve_num => 20,
num_iors => 0,
iors => (
  others => (
    num_chains => 0,
    chains => (
      others => (E=>false,D=>false,C=>false,B=>false,A=>false))))),
7 => (
  gcid => gcids.appl2_3,
  gtid => gtids.appl2,
  location => config.one_vg,
  vg => 0,
  rg => config.rg2,
  precedence => 4,
  max_xmit_size => 200,
  max_xmit_num => 10,
  max_rcve_size => 200,
  max_rcve_num => 20,
  num_iors => 0,
  iors => (
    others => (
      num_chains => 0,
      chains => (
        others => (E=>false,D=>false,C=>false,B=>false,A=>false))))),
8 => (
  gcid => gcids.appl1_4,
  gtid => gtids.appl1,
  location => config.one_vg,
  vg => 0,
  rg => config.rg1,
  precedence => 4,
  max_xmit_size => 200,
  max_xmit_num => 10,
  max_rcve_size => 200,
  max_rcve_num => 20,
  num_iors => 0,
  iors => (
    others => (
      num_chains => 0,
      chains => (
        others => (E=>false,D=>false,C=>false,B=>false,A=>false))))),
9 => (
  gcid => gcids.appl2_4,
  gtid => gtids.appl2,
  location => config.one_vg,
  vg => 0,
  rg => config.rg1,
  precedence => 0,
  max_xmit_size => 200,
  max_xmit_num => 10,
  max_rcve_size => 200,
  max_rcve_num => 20,
  num_iors => 0,
  iors => (
    others => (
```

Appendix F

```
    num_chains => 0,
    chains => (
      others => (E=>false,D=>false,C=>false,B=>false,A=>false))))),

10 => (
  gcid => gcids.test_1,
  gtid => gtids.test,
  location => config.ane_vg,
  vg => 0,
  rg => config.rg1,
  precedence => 1,
  max_xmit_size => 200,
  max_xmit_num => 10,
  max_rcve_size => 200,
  max_rcve_num => 20,
  num_iors => 0,
  iors => (
    others => (
      num_chains => 0,
      chains => (
        others => (E=>false,D=>false,C=>false,B=>false,A=>false))))),

end task_list;
-- DEC/CMS REPLACEMENT HISTORY, Element TASK_LIST.ADA
-- *5   17-FEB-1992 10:11:41 FTPP "added iors spec"
-- *4   12-FEB-1992 12:18:19 SAF2234 "moved dispatcher specs within init_cid"
-- *3    8-FEB-1992 09:52:49 FTPP "added io task assignment"
-- *2   27-DEC-1991 09:46:35 SAF2234 "changed precedence, added appl tasks"
-- *1   13-DEC-1991 10:19:11 SAF2234 "specification for task list in mass
memory"
-- DEC/CMS REPLACEMENT HISTORY, Element TASK_LIST.ADA
```


Appendix F

```

--         scheduler."+(rg_dispatcher.io_interval(i),1.0);
--     end if;
-- end loop;
rg_log.rg_log_entry(my_gcid,"APPL1",
--     text_io.put_line("At wfs of APPL1" &
--         config.gcid_t'image(my_gcid) & " " &
--         config.rg_t'image(my_rg));

loop
-- * for loop: max = 61
  for i in xmessage'range loop
    debug_trace.debug_log(16*f7f7*,byte_to_long(i));
    scheduler.wait_for_schedule;
    debug_trace.debug_log(16*f1f1*,byte_to_long(i));
--     text_io.put_line("After wfs of APPL1" &
rg_log.rg_log_entry(my_gcid,"APPL1",
--         "After wfs " &
--         system.unsigned_byte'image(i) & " " &
--         config.gcid_t'image(my_gcid) & " " &
--         config.rg_t'image(my_rg));

    debug_trace.debug_log(16*f2f2*,byte_to_long(i));
    rg_communication.queue_message(
--         my_gcid,
--         my_gcid,
--         xmessage'address,
--         natural(i),
--         xerror,
--         config.my_vg);
--     if rg_communication."/="(xerror,rg_communication.success) then
--         text_io.put_line(
--             rg_communication.transmit_message_status_t'image(xerror));
--     end if;
    debug_trace.debug_log(16*f3f3*,byte_to_long(i));
    for j in rmessage'range loop
      rmessage(j) := 16*FF*;
    end loop;
    size := natural(rmessage'last);
    debug_trace.debug_log(16*f4f4*,byte_to_long(i));
-- * message: max = 128
    rg_communication.retrieve_message(
--         from_cid,
--         my_gcid,
--         rmessage'address,
--         size,
--         rerror,
--         from_vg,
--         class);
--     if rg_communication."/="(rerror,rg_communication.success) then
--         text_io.put_line(
--             rg_communication.receive_message_status_t'image(rerror));
--         null;
    else
    debug_trace.debug_log(16*f5f5*,byte_to_long(i));
    rg_log.rg_log_entry(my_gcid,"APPL1",
--     text_io.put_line("retrieve: from " &
--         "retrieve: from " &
--         config.gcid_t'image(from_cid) & " to " &
--         config.gcid_t'image(my_gcid) & " " &

```


Appendix F

```

        system.unsigned_longword);
function fetch_word is new system.fetch_from_address(
    system.unsigned_word);
function fetch_byte is new system.fetch_from_address(
    system.unsigned_byte);

xmessage : array (system.unsigned_byte range 0..60) of system.unsigned_byte;
xerror : rg_communication.transmit_message_status_t;

rmessage : array (system.unsigned_byte range 0..60) of system.unsigned_byte;
rerror : rg_communication.receive_message_status_t;
from_cid : config.gcid_t;
from_vg : ne_interface.vgid_t;
size : natural;
class : ne_interface.class_r;

begin
    text_io.put_line("Elaboration of APPL2");
    for i in xmessage'range loop
        xmessage(i) := i;
    end loop;
    accept start(gcid : config.gcid_t) do
        my_gcid := gcid;
    end start;
    my_rg := config.gcid_config(my_gcid).rg;
    rg_log.rg_log_entry(my_gcid, "APPL2",
--    text_io.put_line("At wfs of APPL2" &
        config.gcid_t'image(my_gcid) & " " &
        config.rg_t'image(my_rg));

    loop
-- * for loop: max = 61
        for i in xmessage'range loop
            scheduler.wait_for_schedule;
--            text_io.put_line("After wfs of APPL2" &
--                config.gcid_t'image(my_gcid) & " " &
--                config.rg_t'image(my_rg));

            rg_communication.send_message(
                my_gcid,
                my_gcid,
                xmessage'address,
                natural(i),
                xerror,
                config.my_vg);
--            if rg_communication."/="(xerror,rg_communication.success) then
--                text_io.put_line(
--                    rg_communication.transmit_message_status_t'image(xerror));
--            end if;

            for j in rmessage'range loop
                rmessage(j) := 16*FF#;
            end loop;
            size := natural(rmessage'last);
            rg_communication.read_message(
                from_cid,
                my_gcid,
                rmessage'address,

```


Appendix F

```
-- presence test messages for each virtual group in the configuration.
-- The timeout period is computed as 3 times the frame time of the FDIR task
-- on the tested virtual group. Since FDI task executes as a rate group 4
-- task this is essentially 3 * the minor frame duration.
--
-- When using the NE simulator this time must be exaggerated because the
-- system timestamps are accurate and, of course, the ne sim is not!
```

```
-----
Procedure Compute_UG_Timeout is
```

```
    frame_time_int :      system.unsigned_longword;
    to_int :             system.unsigned_longword;
    to_scale_factor :    system.unsigned_longword := 3;
    to_tick :           clock_extension.system_tick_t;
```

```
begin
```

```
    for i in ne_interface.vgid_t loop
        if config.vgid_config(i).redundancy /= config.redundancy_level_t(0) then

            frame_time_int := ticks_to_int(clock_extension.rep_to_tick
                (config.vgid_config(i).frame));
            to_int := to_scale_factor * frame_time_int;
            to_tick := int_to_ticks(to_int);
```

```
            vgid_status(i).inter_vg_to := clock_extension.systick_to_time(to_tick);
```

```
            vgid_status(i).last_inter_vg_time := initial_time;
            vgid_status(i).cur_inter_vg_time := initial_time;
```

```
        end if;
    end loop;
end;
```

```
-----
-- Read_all_messages reads all messages and updates the information relevant
-- to the specific type of message.
```

```
-- Inter_vg_presence_test messages are used for 2 purposes:
-- 1) the syndrome associated with the message is used in syndrome
-- analysis
-- 2) The time of receipt is used to determine the next expected
-- arrival of an inter_vg_presence_test message. Failure to
-- receive this message within the allotted time implies the
-- UG is faulty.
```

```
-- Syndrome_exchange message contains the syndrome data for a single
-- member of the system FDI UG. The receipt of source congruent messages
-- from each member of the system fdi UG is necessary to perform
-- syndrome analysis. Should one member of the system fdi UG be faulty
-- its data may be corrupted. Consequently, if a source congruent
-- message is received from the system fdi UG, it will be assumed that
-- that message is a syndrome exchange message.
```

```
-----
Procedure Read_all_messages is
```

```
-----
    procedure extract_inter_vg_pt_info(i : in out pt_entry_t) is
    begin
        i := i + 1;
```

Appendix F

```

    pt_buffer(i).from_vg := from_vg;
    pt_buffer(i).syndrome := syndrome;
    pt_buffer(i).class   := class;

    -- Update time of receipt of current inter_vg_presence_test
    -- message
    vgid_status(from_vg).cur_inter_vg_time := syndrome.stamp;

end;
-----
procedure extract_syndrome_exch_info is
begin
    null;
end;
-----
procedure check_for_missing_syndrome_exch is
begin
    null;
--   if from_vg = my_vg then
--       case class is
--           when A =>
--           when B =>
--           when C =>
--           when D =>
--           when E =>
--           when others => software error!!!!
--       end;
--   end if;
end;
-----
begin
    num_pt_entries := 0;
-- * basic loop: max = 12
read_loop:
loop
    -- Receive the inter-UG presence test message from local FDI
    size := rmessage'size/8;
    rg_communication.retrieve_message(
        from_gcid,
        my_gcid,
        rmessage'address,
        size,
        rstatus,
        from_vg,
        class,
        syndrome);
    case rstatus is
        when rg_communication.success =>
            case rmessage.message_type is
                when fdi_msg.INTER_UG_PT =>
                    extract_inter_vg_pt_info(num_pt_entries);
                when fdi_msg.SYNDROME_EXCH =>
                    extract_syndrome_exch_info;
                when others => check_for_missing_syndrome_exch;
            end case;

        when rg_communication.no_message =>
            exit read_loop;
    end case;
end loop;

```

Appendix F

```
        when others =>
--          text_io.put_line(
--            rg_communication.receive_message_status_t'image(rstatus));
            exit read_loop;
        end case;
    end loop read_loop;
```

```
end;
```

```
-----
-- This procedure checks the timeouts of all active UGs. If the time has
-- elapsed beyond the time expected for receipt of a message, the UG will
-- be diagnosed as faulty.
-----
```

```
Procedure Check_inter_vg_timeouts is
```

```
    delta_sys_ticks : clock_extension.system_tick_t;
begin
```

```
    for i in ne_interface.vgid_t loop
        if config.vgid_config(i).redundancy /= config.redundancy_level_t(0) then
```

```
            rg_log.rg_log_entry(my_gcid, "SYSTEM_FDI",
                ne_interface.vgid_t'image(i) &
                clock_extension.system_tick_t'image
                    (vgid_status(i).last_inter_vg_time) & " " &
                clock_extension.system_tick_t'image
                    (vgid_status(i).cur_inter_vg_time));
```

```
            -- Check the next_inter_vg_time to determine if it has expired
            delta_sys_ticks :=
                clock_extension.diff_sys_ticks( vgid_status(i).cur_inter_vg_time,
                                                vgid_status(i).last_inter_vg_time);
            if clock_extension.systick_to_time(delta_sys_ticks) >
                vgid_status(i).inter_vg_to then
                if vgid_status(i).last_inter_vg_time /= initial_time then
                    fdi_log.fdi_log_entry("SYSTEM_FDI",
                        "UG " & ne_interface.vgid_t'image(i) &
                        " failed Inter-UG timeout: " &
                        clock_extension.system_tick_t'image
                            (vgid_status(i).last_inter_vg_time) & " " &
                        clock_extension.system_tick_t'image
                            (vgid_status(i).cur_inter_vg_time));
```

```
                    end if;
                end if;
                vgid_status(i).last_inter_vg_time := vgid_status(i).cur_inter_vg_time;
            end if;
        end loop;
    end;
```

```
-----
Procedure Analyze_syndrome is
```

```
--
-- This procedure analyzes the syndrome matrix received as a result of last
-- frame's syndrome exchange. The syndrome matrix contains the syndrome
-- patterns perceived by each member of the system fdi UG for each UG from
-- whom it received an inter_vg_presence_test message.
```

Appendix F

```
--  
  
begin  
    null;  
  
end Analyze_syndrome;  
  
-----  
Procedure Exchange_syndrome is  
--  
-- This procedure performs a series of source congruent exchanges to  
-- distribute the syndrome data for those inter UG presence test messages  
-- received this iteration. Upon completion each member of the system FDI UG  
-- will have congruent copies of the syndrome bytes.  
--  
begin  
    null;  
  
end exchange_syndrome;  
-----  
  
begin  
    text_io.put_line("Elaboration of SYSTEM_FDI");  
    accept start(gcid : config.gcid_t) do  
        my_gcid := gcid;  
    end start;  
  
    my_rg := config.gcid_config(my_gcid).rg;  
    rg_log.rg_log_entry(my_gcid,"SYSTEM_FDI",  
        config.gcid_t'image(my_gcid) & " " &  
        config.rg_t'image(my_rg));  
  
    -- Compute timeout times for each virtual group in the configuration  
    compute_vg_timeout;  
  
    loop  
  
        scheduler.wait_for_schedule;  
  
        -- Read any messages sent to System FDI  
        Read_all_messages;  
        rg_log.rg_log_entry(my_gcid,"SYSTEM_FDI",  
            pt_entry_t'image(num_pt_entries) & " messages read ");  
  
        -- Examine inter-vg presence test for timeouts  
        Check_inter_vg_timeouts;  
  
        -- Analyze syndrome received this frame  
        Analyze_syndrome;  
  
        -- Exchange syndrome for next frame  
        Exchange_syndrome;  
  
    end loop;  
    exception  
        when NUMERIC_ERROR =>  
            exception_log.exception_log_entry("SYSTEM_FDI"," NUMERIC_ERROR ");  
        when CONSTRAINT_ERROR =>
```


test_code.ada

```

with first_package;
with second_package;
package body test_code is
task body test_t is
begin
    loop
        first_package.second;
        second_package.first;
    end loop;
end test_t;
end test_code;

```

first_package.ada

```

package body first_package is
procedure first is
    i : natural;
    size : natural;
    condition : natural;
    a : natural;
    b : natural;
    c : natural;
    d : natural;
    e : natural;
begin
    for i in 1..2 loop
        rg_communication.retrieve_message(a,b,c,190,d,e);
    end loop;
    -- * while loop: max = 2
    while condition < 10 loop
        rg_communication.queue_message (a,b,c,50,d,e);
        scheduler.wait_for_schedule;
    end loop;
    if condition > 20
        -- * message: max = 50
        rg_communication.queue_message (a,b,c,size,d,e);
    else
        -- * message: max = 100
        rg_communication.queue_message (a,b,c,size,d,e);
    end if;
end first;
procedure second is
    i : natural;
    temperature : natural;
begin
    -- * for loop: max = 11
    for i in 1..natural loop
        first;
        scheduler.wait_for_schedule;
        if i > 6 first;
        end if;
    end loop;
end second;
end first_package;

```

second_package.ada

```
package body second_package is
  procedure first is
    a : natural;
    b : natural;
    c : natural;
    d : natural;
    e : natural;
  begin
    rg_communication.send_message(a,b,c,200,d,e);
  end first;
end second_package;
```


task_names.dat

```
appl1_t  
appl2_t  
system_fdi_t  
test_t  
done
```

list_of_tasks.dat

```
appl1_t 0 4 400 5 400 20  
appl2_t 0 4 200 10 200 20  
system_fdi_t 0 4 200 10 200 20  
appl1_t 0 3 200 10 200 20  
appl2_t 0 3 200 10 200 20  
appl1_t 0 2 400 5 200 20  
appl2_t 0 2 200 10 200 20  
appl1_t 0 1 200 10 200 20  
appl2_t 0 1 200 10 200 20  
test_t 0 1 200 10 200 20
```

filenames.dat

```
appl1_t  
USER:[TREADWELL.AFTA]APP_TEST.ADA;3  
appl2_t  
USER:[TREADWELL.AFTA]APP_TEST.ADA;3  
system_fdi_t  
USER:[TREADWELL.AFTA]SYS_FDI.ADA;12  
test_t  
USER:[TREADWELL.AFTA]TEST_CODE.ADA;6
```

Appendix F

results.dat

All results are given in terms of microseconds.
The allotted minor frame time is 10000.

RESULTS for UG*0

```
TASK: appl1_t  RATE GROUP: 4
WORST CASE PATH: number of packets queued: 7
                  number of messages queued: 1
                  number of packets retrieved: 3
                  number of messages retrieved: 1
                  number of packets sent: 0
                  number of packets read: 0
                  minimal delay: 608

TASK: appl2_t  RATE GROUP: 4
WORST CASE PATH: number of packets queued: 0
                  number of messages queued: 0
                  number of packets retrieved: 0
                  number of messages retrieved: 0
                  number of packets sent: 4
                  number of packets read: 4
                  minimal delay: 534

TASK: system_fdi_t  RATE GROUP: 4
WORST CASE PATH: number of packets queued: 0
                  number of messages queued: 0
                  number of packets retrieved: 48
                  number of messages retrieved: 12
                  number of packets sent: 0
                  number of packets read: 0
                  minimal delay: 3732

TASK: appl1_t  RATE GROUP: 3
WORST CASE PATH: number of packets queued: 4
                  number of messages queued: 1
                  number of packets retrieved: 3
                  number of messages retrieved: 1
                  number of packets sent: 0
                  number of packets read: 0
                  minimal delay: 473

TASK: appl2_t  RATE GROUP: 3
WORST CASE PATH: number of packets queued: 0
                  number of messages queued: 0
                  number of packets retrieved: 0
                  number of messages retrieved: 0
                  number of packets sent: 4
                  number of packets read: 4
                  minimal delay: 534

TASK: appl1_t  RATE GROUP: 2
WORST CASE PATH: number of packets queued: 7
                  number of messages queued: 1
                  number of packets retrieved: 3
                  number of messages retrieved: 1
                  number of packets sent: 0
                  number of packets read: 0
                  minimal delay: 608

TASK: appl2_t  RATE GROUP: 2
WORST CASE PATH: number of packets queued: 0
                  number of messages queued: 0
                  number of packets retrieved: 0
```

Appendix F

```

        number of messages retrieved: 0
        number of packets sent: 4
        number of packets read: 4
        minimal delay: 534
TASK: appl1_t  RATE GROUP: 1
WORST CASE PATH: number of packets queued: 4
                  number of messages queued: 1
                  number of packets retrieved: 3
                  number of messages retrieved: 1
                  number of packets sent: 0
                  number of packets read: 0
                  minimal delay: 473
TASK: appl2_t  RATE GROUP: 1
WORST CASE PATH: number of packets queued: 0
                  number of messages queued: 0
                  number of packets retrieved: 0
                  number of messages retrieved: 0
                  number of packets sent: 4
                  number of packets read: 4
                  minimal delay: 534
TASK: test_t   RATE GROUP: 1
WORST CASE PATH: number of packets queued: 6
                  number of messages queued: 4
                  number of packets retrieved: 16
                  number of messages retrieved: 4
                  number of packets sent: 4
                  number of packets read: 0
                  minimal delay: 1909

```

```

RATE GROUP TOTALS FOR APPLICATION TASKS
rate group 1: 2916
rate group 2: 1142
rate group 3: 1007
rate group 4: 4874

```

```

OVERHEAD TOTALS
minor frame 0: 7041
minor frame 1: 2043
minor frame 2: 2992
minor frame 3: 2081
minor frame 4: 4678
minor frame 5: 2043
minor frame 6: 2992
minor frame 7: 2138

```

MINOR FRAME	OVERHEAD	RG4	RG3	RG2	RG1
0	7041	2959	0	0	0
1	2043	4874	1007	1142	934
2	2992	4874	1007	0	1127
3	2081	4874	0	0	855
4	4678	4874	448	0	0
5	2043	4874	559	1142	0
6	2992	4874	1007	0	0
7	2138	4874	0	0	0

RG4 did not satisfy its boundary in frame 0.

errors.dat

The matching between tasks and filenames is...
 0 appl1_t USER:[TREADWELL.AFTA]APP_TEST.ADA;3
 1 appl2_t USER:[TREADWELL.AFTA]APP_TEST.ADA;3
 2 system_fdi_t USER:[TREADWELL.AFTA]SYS_FDI.ADA;12
 3 test_t USER:[TREADWELL.AFTA]TEST_CODE.ADA;6

SOFTWARE ANALYSIS FOR appl1_t

The packages found are...none
 Now processing task appl1_t
 Assuming default size for queue_message
 Task Model...
 0 Type: 1 Value: -3 Depth: 0 Pointer: 7
 1 Type: 55 Value: 61 Depth: 1 Pointer: 6
 2 Type: 0 Value: -2 Depth: 2 Pointer: -2
 3 Type: 7 Value: 7 Depth: 2 Pointer: -2
 4 Type: 8 Value: 3 Depth: 2 Pointer: -2
 5 Type: 51 Value: -2 Depth: 1 Pointer: 1
 6 Type: 51 Value: -2 Depth: 0 Pointer: 0
 PATH: 0 1 2
 PATH: 3 4 5 1 2
 PATH: 3 4 5 6 0 1 2
 The worst path is characterized by...
 Queued: 7 Retrieved: 3 Sent: 0 Read: 0

SOFTWARE ANALYSIS FOR appl2_t

The packages found are...none
 Now processing task appl2_t
 Unexpected task name -- found: appl1_t expected: appl2_t
 Assuming default size for queue_message
 Assuming default size for send_message
 Assuming default size for read_message
 Task Model...
 0 Type: 1 Value: -3 Depth: 0 Pointer: 7
 1 Type: 55 Value: 61 Depth: 1 Pointer: 6
 2 Type: 0 Value: -2 Depth: 2 Pointer: -2
 3 Type: 9 Value: 4 Depth: 2 Pointer: -2
 4 Type: 10 Value: 4 Depth: 2 Pointer: -2
 5 Type: 51 Value: -2 Depth: 1 Pointer: 1
 6 Type: 51 Value: -2 Depth: 0 Pointer: 0
 PATH: 0 1 2
 PATH: 3 4 5 1 2
 PATH: 3 4 5 6 0 1 2
 The worst path is characterized by...
 Queued: 0 Retrieved: 0 Sent: 4 Read: 4

Appendix F

SOFTWARE ANALYSIS FOR system_fdi_t

```
The packages found are...none
Now processing task system_fdi_t
Assuming default size for retrieve_message
Procedure compute_vg_timeout found in USER:[TREADWELL.AFTA]SYS_FDI.ADA;12
Type: 55 Value: -2 Depth: 0 Pointer: 2
Type: 51 Value: -2 Depth: 0 Pointer: 0
Procedure read_all_messages found in USER:[TREADWELL.AFTA]SYS_FDI.ADA;12
Type: 54 Value: 0 Depth: 0 Pointer: -2
Ctr 0: Queued 0 Rtrvd 48 Sent 0 Read 0
Procedure extract_inter_vg_pt_info found in USER:[TREADWELL.AFTA]SYS_FDI.ADA;12
Procedure extract_syndrome_exch_info found in USER:[TREADWELL.AFTA]SYS_FDI.ADA;12
Procedure check_for_missing_syndrome_exch found in
USER:[TREADWELL.AFTA]SYS_FDI.ADA;12
Procedure check_inter_vg_timeouts found in USER:[TREADWELL.AFTA]SYS_FDI.ADA;12
Type: 55 Value: -2 Depth: 0 Pointer: 2
Type: 51 Value: -2 Depth: 0 Pointer: 0
Procedure analyze_syndrome found in USER:[TREADWELL.AFTA]SYS_FDI.ADA;12
Procedure exchange_syndrome found in USER:[TREADWELL.AFTA]SYS_FDI.ADA;12
Task Model...
  0 Type: 1 Value: -3 Depth: 0 Pointer: 3
  1 Type: 0 Value: -2 Depth: 1 Pointer: -2
  2 Type: 54 Value: 0 Depth: 1 Pointer: -2
  3 Type: 51 Value: -2 Depth: 0 Pointer: 0
Ctr 0: Queued 0 Retvd 48 Sent 0 Read 0
PATH: 0 1
PATH: 2 3 0 1
The worst path is characterized by...
Queued: 0 Retrieved: 48 Sent: 0 Read: 0
```

SOFTWARE ANALYSIS FOR appl1_t

```
The packages found are...none
Now processing task appl1_t
Assuming default size for queue_message
Task Model...
  0 Type: 1 Value: -3 Depth: 0 Pointer: 7
  1 Type: 55 Value: 61 Depth: 1 Pointer: 6
  2 Type: 0 Value: -2 Depth: 2 Pointer: -2
  3 Type: 7 Value: 4 Depth: 2 Pointer: -2
  4 Type: 8 Value: 3 Depth: 2 Pointer: -2
  5 Type: 51 Value: -2 Depth: 1 Pointer: 1
  6 Type: 51 Value: -2 Depth: 0 Pointer: 0
PATH: 0 1 2
PATH: 3 4 5 1 2
PATH: 3 4 5 6 0 1 2
The worst path is characterized by...
Queued: 4 Retrieved: 3 Sent: 0 Read: 0
```

Appendix F

SOFTWARE ANALYSIS FOR appl2_t

The packages found are...none

Now processing task appl2_t

Unexpected task name -- found: appl1_t expected: appl2_t

Assuming default size for queue_message

Assuming default size for send_message

Assuming default size for read_message

Task Model...

0	Type:	1	Value:	-3	Depth:	0	Pointer:	7
1	Type:	55	Value:	61	Depth:	1	Pointer:	6
2	Type:	0	Value:	-2	Depth:	2	Pointer:	-2
3	Type:	9	Value:	4	Depth:	2	Pointer:	-2
4	Type:	10	Value:	4	Depth:	2	Pointer:	-2
5	Type:	51	Value:	-2	Depth:	1	Pointer:	1
6	Type:	51	Value:	-2	Depth:	0	Pointer:	0

PATH: 0 1 2

PATH: 3 4 5 1 2

PATH: 3 4 5 6 0 1 2

The worst path is characterized by...

Queued: 0 Retrieved: 0 Sent: 4 Read: 4

SOFTWARE ANALYSIS FOR appl1_t

The packages found are...none

Now processing task appl1_t

Assuming default size for queue_message

Task Model...

0	Type:	1	Value:	-3	Depth:	0	Pointer:	7
1	Type:	55	Value:	61	Depth:	1	Pointer:	6
2	Type:	0	Value:	-2	Depth:	2	Pointer:	-2
3	Type:	7	Value:	7	Depth:	2	Pointer:	-2
4	Type:	8	Value:	3	Depth:	2	Pointer:	-2
5	Type:	51	Value:	-2	Depth:	1	Pointer:	1
6	Type:	51	Value:	-2	Depth:	0	Pointer:	0

PATH: 0 1 2

PATH: 3 4 5 1 2

PATH: 3 4 5 6 0 1 2

The worst path is characterized by...

Queued: 7 Retrieved: 3 Sent: 0 Read: 0

Appendix F

SOFTWARE ANALYSIS FOR appl2_t

```
The packages found are...none
Now processing task appl2_t
Unexpected task name -- found: appl1_t expected: appl2_t
Assuming default size for queue_message
Assuming default size for send_message
Assuming default size for read_message
Task Model...
  0 Type: 1 Value: -3 Depth: 0 Pointer: 7
  1 Type: 55 Value: 61 Depth: 1 Pointer: 6
  2 Type: 0 Value: -2 Depth: 2 Pointer: -2
  3 Type: 9 Value: 4 Depth: 2 Pointer: -2
  4 Type: 10 Value: 4 Depth: 2 Pointer: -2
  5 Type: 51 Value: -2 Depth: 1 Pointer: 1
  6 Type: 51 Value: -2 Depth: 0 Pointer: 0
PATH: 0 1 2
PATH: 3 4 5 1 2
PATH: 3 4 5 6 0 1 2
The worst path is characterized by...
Queued: 0 Retrieved: 0 Sent: 4 Read: 4
```

SOFTWARE ANALYSIS FOR appl1_t

```
The packages found are...none
Now processing task appl1_t
Assuming default size for queue_message
Task Model...
  0 Type: 1 Value: -3 Depth: 0 Pointer: 7
  1 Type: 55 Value: 61 Depth: 1 Pointer: 6
  2 Type: 0 Value: -2 Depth: 2 Pointer: -2
  3 Type: 7 Value: 4 Depth: 2 Pointer: -2
  4 Type: 8 Value: 3 Depth: 2 Pointer: -2
  5 Type: 51 Value: -2 Depth: 1 Pointer: 1
  6 Type: 51 Value: -2 Depth: 0 Pointer: 0
PATH: 0 1 2
PATH: 3 4 5 1 2
PATH: 3 4 5 6 0 1 2
The worst path is characterized by...
Queued: 4 Retrieved: 3 Sent: 0 Read: 0
```

Appendix F

SOFTWARE ANALYSIS FOR appl2_t

The packages found are...none

Now processing task appl2_t

Unexpected task name -- found: appl1_t expected: appl2_t

Assuming default size for queue_message

Assuming default size for send_message

Assuming default size for read_message

Task Model...

0 Type: 1 Value: -3 Depth: 0 Pointer: 7

1 Type: 55 Value: 61 Depth: 1 Pointer: 6

2 Type: 0 Value: -2 Depth: 2 Pointer: -2

3 Type: 9 Value: 4 Depth: 2 Pointer: -2

4 Type: 10 Value: 4 Depth: 2 Pointer: -2

5 Type: 51 Value: -2 Depth: 1 Pointer: 1

6 Type: 51 Value: -2 Depth: 0 Pointer: 0

PATH: 0 1 2

PATH: 3 4 5 1 2

PATH: 3 4 5 6 0 1 2

The worst path is characterized by...

Queued: 0 Retrieved: 0 Sent: 4 Read: 4

SOFTWARE ANALYSIS FOR test_t

The packages found are...

first_package

second_package

Now processing package second_package

Now processing package first_package

Now processing task test_t

Procedure second_package.first found in second_package.ada

Type: 9 Value: 4 Depth: 0 Pointer: -2

Procedure first_package.first found in first_package.ada

Type: 54 Value: 0 Depth: 0 Pointer: -2

Type: 56 Value: 2 Depth: 0 Pointer: 4

Type: 7 Value: 1 Depth: 1 Pointer: -2

Type: 0 Value: -2 Depth: 1 Pointer: -2

Type: 51 Value: -2 Depth: 0 Pointer: 1

Type: 54 Value: 1 Depth: 0 Pointer: -2

Ctr 0: Queued 0 Rtrvd 8 Sent 0 Read 0

Ctr 1: Queued 3 Rtrvd 0 Sent 0 Read 0

Procedure first_package.second found in first_package.ada

Type: 55 Value: 11 Depth: 0 Pointer: 16

Type: 54 Value: 0 Depth: 1 Pointer: -2

Type: 56 Value: 2 Depth: 1 Pointer: 5

Type: 7 Value: 1 Depth: 2 Pointer: -2

Type: 0 Value: -2 Depth: 2 Pointer: -2

Type: 51 Value: -2 Depth: 1 Pointer: 2

Type: 54 Value: 1 Depth: 1 Pointer: -2

Type: 0 Value: -2 Depth: 1 Pointer: -2

Type: 2 Value: -2 Depth: 1 Pointer: 15

Type: 54 Value: 2 Depth: 2 Pointer: -2

Type: 56 Value: 2 Depth: 2 Pointer: 13

Type: 7 Value: 1 Depth: 3 Pointer: -2

Type: 0 Value: -2 Depth: 3 Pointer: -2

Type: 51 Value: -2 Depth: 2 Pointer: 10

Type: 54 Value: 3 Depth: 2 Pointer: -2

Type: 52 Value: -2 Depth: 1 Pointer: 8

Appendix F

```

Type: 51 Value: -2 Depth: 0 Pointer: 0
Ctr 0: Queued 0 Rtrvd 8 Sent 0 Read 0
Ctr 1: Queued 3 Rtrvd 0 Sent 0 Read 0
Ctr 2: Queued 0 Rtrvd 8 Sent 0 Read 0
Ctr 3: Queued 3 Rtrvd 0 Sent 0 Read 0
Task Model...
 0 Type: 1 Value: -3 Depth: 0 Pointer: 19
 1 Type: 55 Value: 11 Depth: 1 Pointer: 17
 2 Type: 54 Value: 0 Depth: 2 Pointer: -2
 3 Type: 56 Value: 2 Depth: 2 Pointer: 6
 4 Type: 7 Value: 1 Depth: 3 Pointer: -2
 5 Type: 0 Value: -2 Depth: 3 Pointer: -2
 6 Type: 51 Value: -2 Depth: 2 Pointer: 3
 7 Type: 54 Value: 1 Depth: 2 Pointer: -2
 8 Type: 0 Value: -2 Depth: 2 Pointer: -2
 9 Type: 2 Value: -2 Depth: 2 Pointer: 16
10 Type: 54 Value: 2 Depth: 3 Pointer: -2
11 Type: 56 Value: 2 Depth: 3 Pointer: 14
12 Type: 7 Value: 1 Depth: 4 Pointer: -2
13 Type: 0 Value: -2 Depth: 4 Pointer: -2
14 Type: 51 Value: -2 Depth: 3 Pointer: 11
15 Type: 54 Value: 3 Depth: 3 Pointer: -2
16 Type: 52 Value: -2 Depth: 2 Pointer: 9
17 Type: 51 Value: -2 Depth: 1 Pointer: 1
18 Type: 9 Value: 4 Depth: 1 Pointer: -2
19 Type: 51 Value: -2 Depth: 0 Pointer: 0
Ctr 0: Queued 0 Retvd 8 Sent 0 Read 0
Ctr 1: Queued 3 Retvd 0 Sent 0 Read 0
Ctr 2: Queued 0 Retvd 8 Sent 0 Read 0
Ctr 3: Queued 3 Retvd 0 Sent 0 Read 0
PATH: 0 1 2 7 8
PATH: 0 1 2 3 4 5
PATH: 6 7 8
PATH: 6 3 4 5
PATH: 6 7 8
PATH: 16 17 1 2 7 8
PATH: 16 17 1 2 3 4 5
PATH: 16 17 18 19 0 1 2 7 8
PATH: 16 17 18 19 0 1 2 3 4 5
PATH: 9 10 15 16 17 1 2 7 8
PATH: 9 10 15 16 17 1 2 3 4 5
PATH: 9 10 15 16 17 18 19 0 1 2 7 8
PATH: 9 10 15 16 17 18 19 0 1 2 3 4 5
PATH: 9 10 11 12 13
PATH: 14 15 16 17 1 2 7 8
PATH: 14 15 16 17 1 2 3 4 5
PATH: 14 15 16 17 18 19 0 1 2 7 8
PATH: 14 15 16 17 18 19 0 1 2 3 4 5
PATH: 14 11 12 13
PATH: 14 15 16 17 1 2 7 8
PATH: 14 15 16 17 1 2 3 4 5
PATH: 14 15 16 17 18 19 0 1 2 7 8
PATH: 14 15 16 17 18 19 0 1 2 3 4 5
The worst path is characterized by...
Queued: 6 Retrieved: 16 Sent: 4 Read: 0

```

Appendix G

A Checklist for Adding Critical Constructs

1. Designate a constant name that is to be used to refer to the new construct and add the constant definition to the list in `HEADER.H`

EX: `define NEW_CONSTRUCT 28`

2. Add a new integer parameter to the `counter_set` structure definition in `HEADER.H`.

EX: `int new_construct;`

3. Add the new delay parameter to the `constant_list` structure definition listed in `HEADER.H`.

EX: `int new_construct_const;`

4. Add the new construct to the end of the list in “`key_words.dat.`”

5. Add the delay constant corresponding to the new construct to “`constants.dat.`”

EX: `new_construct_delay 33`

6. Add the new delay parameter to the `read_list` procedure for reading in delay constants.

EX: `fscanf(list_file, "%s %d\n", dummy_string, &dummy_int);
delay_data->new_construct_const = dummy_int;`

7. Add the new parameter to the initialization lists found in `task_parse`, `generate_paths`, and `find_worst_path`.

EX: `final_counter.new_construct = 0;`

8. Add the new construct to the switch statement in the `search` procedure. If it is a subprogram call, group it with the message passing calls and use the `valid_call` procedure to verify any occurrence of the new construct. The code should look something like this...

EX: `case NEW_CONSTRUCT:
 if(valid_call(this_line->entry[i].name,
 search_list->entry[j].name) == YES)
 {
 skeleton->entry[skeleton->length].type = NEW_CONSTRUCT;
 skeleton->entry[skeleton->length].value = UNDEFINED;
 if (flags->ctr_active) ++skeleton->length;
 }
 break;`

9. Add the new construct to the summation in `check_ctrs`.

10. Add the new construct to the switch statement in `parameterize`.

EX: `case NEW_CONSTRUCT:
 ++counter_set.new_construct;
 break;`

Appendix G

11. Add the new construct to the calculations in `calculate_time`.

EX: `sum += delay_data->new_construct_const*counter_set.new_construct;`

12. Add the new construct to the parameter listing in `write_file`.

EX: `fprintf(outfile, " new construct instances: %d\n",
ug[i].task[j].counter_set.new_construct);`

Appendix H

References

- [AFTA91] Harper R. et. al. *The Army Fault Tolerant Architecture Conceptual Study*, prepared for US Army AVRADA, Ft. Monmouth N.J., Aug. 1991.
- [BOO87] Booch, Grady. *Software Engineering With Ada*. Menlo Park CA: The Benjamin Cummings Publishing Company, Inc., 1987.
- [CLAS92] Clasen, R. et. al. "Empirical Performance of a Fault-Tolerant Hard-Real-Time Parallel Processor," Oct. 1992.
- [CLAS93] Clasen, R. *Performance Modeling and Analysis of a Fault-Tolerant, Real-Time Parallel Processor*, Master of Science Thesis, Northeastern University, June 1993.
- [DOL82] Dolev, D. "The Byzantine Generals Strike Again," *Journal of Algorithms*, vol. 3, 1982, pp. 14-30.
- [DOL84] Dolev, D. et. al. "Fault Tolerant Clock Synchronization," *Communications of ACM*, 1984, pp. 89-101.
- [HAR91] Harper, R. E. and J. H. Lala. "Fault-Tolerant Parallel Processor," *Journal of Guidance, Control, and Dynamics*, vol 14, no. 3, May 1991, pp. 554-563.
- [LAM82] Lamport, L. et. al. "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, Jul 1982, pp. 383-401.
- [PARK90] Park, ChangYun and Alan C. Shaw. "Experiments With a Program Timing Tool Based on Source-Level Timing Schema," *Proc. 11th IEEE Real-Time Systems Symposium*, Dec. 1990, pp. 72-81.
- [PUS89] Puschner P. and Ch. Koza. "Calculating the Maximum Execution Time of Real-Time Programs," *The Journal of Real-Time Systems*, Sep 1989, 1(2):159-176.