INFORMATION TO USERS

While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. For example:

- Manuscript pages may have indistinct print. In such cases, the best available copy has been filmed.

- Manuscripts may not always be complete. In such cases, a note will indicate that it is not possible to obtain missing pages.

- Copyrighted material may have been removed from the manuscript. In such cases, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or as a 17”x 23” black and white photographic print.

Most photographs reproduce acceptably on positive microfilm or microfiche but lack the clarity on xerographic copies made from the microfilm. For an additional charge, 35mm slides of 6”x 9” black and white photographic prints are available for any photographs or illustrations that cannot be reproduced satisfactorily by xerography.
Adapting real-time software for reliable performance

Bihari, Thomas Edward, Ph.D.

The Ohio State University, 1987

Copyright ©1987 by Bihari, Thomas Edward. All rights reserved.
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs or pages
2. Colored illustrations, paper or print
3. Photographs with dark background
4. Illustrations are poor copy
5. Pages with black marks, not original copy
6. Print shows through as there is text on both sides of page
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements
9. Tightly bound copy with print lost in spine
10. Computer printout pages with indistinct print
11. Page(s) lacking when material received, and not available from school or author.
12. Page(s) seem to be missing in numbering only as text follows.
13. Two pages numbered. Text follows.
14. Curling and wrinkled pages
15. Dissertation contains pages with print at a slant, filmed as received
16. Other

University Microfilms International
ADAPTING REAL-TIME SOFTWARE
FOR
RELIABLE PERFORMANCE

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

BY
Thomas Edward Bihari, B.S., M.S., M.S.

* * * * *

The Ohio State University
1987

Dissertation Committee:
Dr. Karsten Schwan
Dr. Bruce Weide
Dr. Fusun Ozguner

Approved By

Adviser
Department of Computer
and Information Science
To My Parents
Acknowledgments

I would like to thank my advisor, Dr. Karsten Schwan, for suggesting the topic of this dissertation and for his considerable help along the way. I would like to thank Dr. Bruce Weide and Dr. Fusun Ozguner for serving on my dissertation reading committee. I would like to thank all of the members of the ISSOS group, and the Adaptive Suspension Vehicle project for their help and support, especially Prabha Gopinath and Win Bo for spending many hours building and maintaining the computer hardware and system software described in this dissertation.
Vita

April 13, 1956  Born - Dover, Ohio

1978  B.S. Mathematics,
      Kent State University,
      Kent, Ohio

1978-1980  Systems Programmer,
           Diebold Inc.
           North Canton, Ohio

1982  M.S. Mathematics,
      The Ohio State University,
      Columbus, Ohio

1983  M.S. Computer and Information Science,
      The Ohio State University,
      Columbus, Ohio

1987  Ph.D. Computer and Information Science,
      The Ohio State University,
      Columbus, Ohio
Publications


Research Interests

Robotics, Parallel Real-Time Systems, Operating Systems, Computer Reliability, Artificial Intelligence pertaining to Robotics, and/or related areas.
# Table of Contents

Acknowledgments iii  
Vita iv  
Table of Contents vi  
List of Figures x  

1. Introduction 1  
1.1 Motivation 1  
1.2 Preliminary Discussion 3  
1.3 An Example 5  
1.4 Adaptations 9  
1.5 Adaptability and Reliable Real-Time Systems 12  
1.6 Automatic Adaptations and Real-Time Systems 15  
1.7 Thesis 16  
1.8 The Range of this Dissertation 16  
1.9 A Real-Time Software Adaptation System 19  
1.10 Overview of the Rest of the Dissertation 22  

2. Reliability in Real-Time Systems 24  
2.1 Real-Time Systems 24  
2.1.1 Definitions 24  
2.1.2 Typical Real-Time Systems 26  
2.2 Reliability in Real-time Systems 29  
2.2.1 Definitions 29  
2.2.2 Reliability Impairments 30  
2.2.3 Reliability Measures 32  
2.2.4 Reliability Means 33  
2.2.4.1 Fault Detection 34  
2.2.4.2 Fault Masking 36  
2.2.4.3 Error Recovery 39  
2.3 Reliability Adaptations for Real-Time Systems 41  

3. Programming Model 43
3.1 The RESAS Programming Model
   3.1.1 Basic Objects
      3.1.1.1 Inter-Object Communication
      3.1.1.2 The Operation of Basic Objects.
      3.1.1.3 Serializability and Concurrency Control.
      3.1.1.4 Input Requirements
      3.1.1.5 Views of Basic Objects
      3.1.1.6 Basic Object Equivalence
      3.1.1.7 Summary
   3.1.2 Composite Objects
      3.1.2.1 Call-Return Communication
      3.1.2.2 Views of Composite Objects.
      3.1.2.3 Naming Components of Composite Objects
      3.1.2.4 Composite Object Equivalence
      3.1.2.5 Composite Object - Basic Object Equivalence
   3.1.3 Concrete and Synthetic Types
   3.2 Discussion
      3.2.1 Adaptability Considerations
      3.2.2 Parallelism Considerations
      3.2.3 Reliability Considerations
      3.2.4 Real-time Considerations
   3.3 Related Research
   3.4 A Restriction of the Object Model
      3.4.1 Input Interfaces
      3.4.2 Real-Time Programs
      3.4.3 Object Scheduling
   3.5 An Example
4. Representation Framework
   4.1 Introduction
   4.2 The Augmented Entity/Relationship Model
      4.2.1 Entities and Relationships
      4.2.2 The Format of Entities and Relationships
      4.2.3 A Note on Syntax
      4.2.4 Operations on Entities
         4.2.4.1 Creation
         4.2.4.2 Attribute Manipulation
      4.2.5 Operations on Relationships
         4.2.5.1 Creation
         4.2.5.2 Attribute Manipulation
         4.2.5.3 Tuples Attribute Manipulation
         4.2.5.4 The Sort Operation
4.2.5.5 The Copy Operation 98
4.2.5.6 Accessing Individual Tuples 99
4.2.5.7 Tuple Manipulation 100
4.2.6 Operations Combining Entities and Relationships 101
4.2.7 Action Routines 103
4.3 Implementation of the Entity/Relationship Framework 105
4.4 Discussion 107
4.5 Related Research 111
4.6 Mapping Objects to ER Model 113
4.6.1 Useful Views 119
5. The Adaptation Control System 121
5.1 The Monitoring Mechanism 123
5.2 The Adaptation Controller 125
5.2.1 Fault Modes 128
5.3 The Adaptation Enactment Mechanism 130
5.3.1 Adaptation Transactions 131
5.3.2 Adaptation Transactions and the DMS 135
6. Implementation of the RESAS Prototype 138
6.1 Introduction 138
6.2 The Hardware 138
6.3 The GEM Operating System 141
6.4 Generating Objects 142
6.4.1 RESAS Objects 142
7. Generation of the Initial Configuration 151
7.1 Introduction 151
7.2 Instantiating Objects 152
7.3 Activations 157
7.4 Instantiating Activations 158
7.5 Timing Analysis 161
7.6 Service Calculation 163
7.7 Processor Assignment 167
7.7.1 The Processor Assignment Algorithm 168
7.7.2 Deadline Calculation 171
7.7.3 Schedule Generation 172
7.7.4 The Processor Choice Algorithms 174
7.7.5 Assigning Multi-Activation Objects 175
7.7.6 Schedule Stretching 178
7.8 An Example 179
8. Iterative Static Adaptation 181
8.1 Introduction 181
List of Figures

Figure 1: Robot Control Software 6
Figure 2: The Prototype Real-Time Software Adaptation System 19
Figure 3: A Typical Real-time Control System 27
Figure 4: A Basic Object 45
Figure 5: A Programmer's View of a Basic Object 53
Figure 6: Call-Return Links 60
Figure 7: A Programmer's View of a Composite Object 62
Figure 8: Composite Objects and Names 65
Figure 9: A Type Hierarchy 68
Figure 10: Control Program 81
Figure 11: Duplicated Force Transformation Type 82
Figure 12: Force Transformation Type Hierarchy 83
Figure 13: The RESAS Adaptation Control System 122
Figure 14: The Prototype RESAS Hardware 139
Figure 15: A Typical RESAS Basic Object 143
Figure 16: Composite and Basic Types 154
Figure 17: Instantiating Object S of Type System 155
Figure 18: Resolving Object S of Type System 156
Figure 19: Activations and Their Invocations 159
Figure 20: Service Values 165
Figure 21: The Inverse Plant Plus Jacobian Control Program 199
Figure 22: Object Execution Time vs. Normalized Object Failures 206
Figure 23: Static Algorithm Tests 211
Figure 24: Dynamic Algorithm Tests 224
Chapter 1
Introduction

1.1 Motivation

To provide increased performance and reliability, embedded multiprocessors and distributed computer systems are replacing uniprocessors and "wired-logic" in many real-time systems. As a result, the reliability of the embedded multiple computer systems has become a critical factor in the overall reliability of such real-time systems. In some cases, high reliability has become important for the physical safety of users of the systems. As the embedded parallel software has taken on more of the tasks necessary for the survival of the enclosing real-time systems, reliability has come to depend on the ability of the software to be adapted to meet changing real-time needs and computer hardware capabilities.
This dissertation examines the problem of adapting software to achieve two goals:

• software execution within specified time constraints, and
• software resiliency with respect to computer hardware failures.

To achieve these goals, a **Real-time Software Adaptation System (RESAS)** has been designed and implemented. It includes:

- an **object-based programming model** which allows the application programmer to create and manipulate adaptable software,

- an **entity/relationship representation framework** which is used to represent important attributes of the software, the computer hardware and their interactions, and

- an **adaptation control system (ACS)** consisting of:
  - a **monitoring mechanism (MON)** which observes and reports runtime information,
  - a **data management system (DMS)** which stores and presents information used by the other RESAS components,
  - an **adaptation controller (AC)** which makes decisions on the appropriate choices of software configurations necessary to meet the stated real-time and reliability goals, and
  - an **adaptation enactment mechanism (AE)** which performs the adaptations chosen by the adaptation controller.

In this dissertation, the theoretical and practical issues associated with software adaptation are discussed, the design and implementation of the RESAS prototype are described, and the results of several tests of RESAS, performing adaptations on a real-time program executing on a multiprocessor, are given.
1.2 Preliminary Discussion

In order to describe RESAS and the notion of adaptations, a few definitions are necessary.

A target system is a combination of software components and hardware components which combine to offer some service to a surrounding environment in which it is embedded.

A particular target system configuration consists of a hardware configuration (e.g. processors and communication channels), a software configuration (e.g. the set of software components and their connections) and the software-hardware map (e.g. which software components are assigned to execute on which processors).

Parallel computer hardware embedded in real-time systems may include tightly-coupled bus-based [44, 14, 77] multiprocessors, interconnection-network-based [16] multiprocessors, message-passing cube topologies [4], serial broadcast bus [73] multicomputers, local area network multicomputers and systolic arrays [40]. Hardware configurations may be heterogeneous, containing special purpose components, such as array processors for FFT calculations and Lisp machines for embedded expert systems, as well as general purpose computers [43].
Software that is changed with the intent of improving performance or reliability is termed adaptable, and the changes to the software are called adaptations. In embedded real-time systems, the software of the target system (also known as the target software or target program) may comprise both the application code and the operating system utilities (e.g. communication and scheduling primitives) employed by that code, and is therefore termed operating software. The adaptation of operating software may therefore include changes to both application code and operating system code.

The term reliability will be used in a general sense: the degree to which reliance can be placed on a system to deliver the expected services. In a real-time computer system, the reliability of the system implies not only that the system performs appropriate actions, but also that those actions are performed within specified timing constraints.

In many cases, software development is performed on development systems attached to the embedded computer hardware. The development systems generally include software development tools, such as editors, compilers and debuggers. Application programmers develop the software using the tools, and the executable software is loaded into the embedded computer hardware and executed.
1.3 An Example

Figure 1 shows a real-time control program [39]. Although real-time programs come in a variety of shapes and sizes, this program exhibits characteristics which are common to many real-time programs.

The software takes as input the current positions and velocities of a robot manipulator’s joints, and the desired positions and velocities of those joints. It then calculates the necessary torque commands to send to the joint actuators in order to achieve the desired positions and velocities.

The software is made up of a number of components, some of which may run in parallel and some of which must run sequentially. The software components have various execution times and the messages passed from component to component are of various sizes. The time (i.e. transport lag) between the collection of new inputs and the generation of the corresponding torques is bounded by a deadline determined by the physical characteristics of the manipulator. If this deadline is missed (or if no outputs, or incorrect outputs, are generated due to failures), the manipulator may become unstable. To ensure that the overall deadline is met, individual components may be assigned “intermediate” deadlines, which may be used as guidelines by the software scheduler.
Figure 1: Robot Control Software
The software is generated on a development system and is executed on a multiprocessor attached to the manipulator. The software life cycle proceeds through the design, implementation and testing stages, and eventually the software is made operational (i.e. it is released).

During the software development stages, programmers use the tools provided by the development system to generate and integrate the software components. The software is loaded into the multiprocessor and its performance and reliability characteristics are measured. These measurements may lead to redesign and remeasurement. Changes may include modification of the assignment of software components to processors, replacement of one version of a component with another version and changes to the intermediate deadlines.

Once the software is released and enters its operational stage, data may continue to be collected from the executing system, and system maintenance personnel may make changes to the software (similar to those mentioned above) as new performance and reliability information becomes available.

Analysis and decision algorithms may be associated with the changes. These algorithms may include, for example, integer programming algorithms for software-to-hardware assignment, and data-flow error
propagation analysis. Algorithms included in the executing software may automatically change the software in response to situations that arise during system operation. For example, a permanent (hard) processor fault may result in other processors taking over the duties of the faulty processor.

The changes described above are performed at various points in the life of the system, and they may involve major changes in the actual implementation, requiring the use of the development system tools, or minor implementation changes which may be carried out by tools executing on the multiprocessor. Nevertheless, all of the changes are directed toward a common goal — increased performance and reliability — and therefore a distinction between the changes, arising from the different mechanisms used to implement them or from the different times at which they take place, is undesirable.

This dissertation proposes a uniform system for generating and adapting software for such real-time systems. The goals of the research described in this dissertation include ease of software experimentation and maintenance, and some degree of autonomous corrective and preventive behavior with respect to real-time and reliability requirements.
1.4 Adaptations

As described in the previous sections, the notion of adaptation is quite general. The study of adaptation encompasses much other research, including processor load-balancing [71], database reorganization in response to changing utilization and failure patterns [66] and dynamic reconfiguration of systolic arrays in response to failures [40]. Adaptations may include:

- Changes in the software configuration, including:
  - Substitutions of software components by others with differing reliability characteristics, such as the substitution of a component that performs little fault detection by one with enhanced fault detection and masking capabilities.
  - Modifications to the software interaction structure, such as the replacement of a request for a service provided by an overloaded component by a request to a more lightly loaded component.
  - Changes to the software interaction characteristics, such as the modification of deadlines for service delivery.

- Changes to the internal structure and function of software components, such as the modification of the maximum number of retries allowed for a component.

- Changes to the software to hardware map, such as the modification of the assignment of software processes to processors.

Adaptations may be performed statically — offline between executions of the target system, or dynamically — on-the-fly during the
execution of the target system. Static adaptations tend to be more "powerful" since the real-time constraints on data analysis, decision-making and adaptation enactment are generally less restrictive than those of dynamic adaptations. Also, some software development tools (e.g. compilers) may be available for use in static adaptations, but not for dynamic adaptations. Dynamic adaptations complement static adaptations by allowing faster, incremental changes to the executing target system, possibly in response to unanticipated events.

The decision to perform an adaptation may be arrived at automatically — based on algorithms triggered by data collected from the executing target system, or by user request — at the request of application programmers or maintenance personnel.

The adaptations themselves may be application-independent — depending on characteristics common to all target software in a particular domain, or application-dependent — depending on specific characteristics of a particular application program. Algorithms associated with application-independent adaptations may be composed by a system designer familiar with the application domain and typical target hardware, and the algorithms may then be used by application programmers. Algorithms associated with application-dependent adaptations must be developed in conjunction with the particular application software.
Adaptability is a desirable attribute of software during several stages in the lifetime of a target system. During the software development stage, adaptations may aid the programmer by allowing experimental programming [4]. That is, different software configurations and software-hardware maps may be built, executed and evaluated. The ability to do fast prototyping of hardware [22] and software [5], has created the need for a software adaptation system which can express such adaptations of the target software and can aid in the software development process.

During the operational life of a target system, adaptations may be performed at the request of system maintenance personnel, or automatically, in response to conditions detected by the monitoring mechanism.

In both the development and operational stages, adaptations may be used in a variety of ways, depending on the types of software components, the capabilities of the target hardware components, the capabilities of the development system and the requirements of the surrounding environment. In time-critical applications, static (and possibly dynamic) adaptations may be performed during the development phase of the project. During operational missions, fast dynamic adaptations may be performed, while static adaptations may be performed between missions.
1.5 Adaptability and Reliable Real-Time Systems

Previous projects have addressed the fact that the reliability, and the reliability requirements, of systems may change over time. For example, the Space Shuttle computer system [19] can be configured in different ways at different stages of a flight, depending on the criticalness of the computation. Similarly, the C.vmp [64] project addressed the issue of dynamic performance-reliability tradeoffs by allowing dynamic voting control. RESAS addresses such performance-reliability tradeoffs. It does so, however, within a general framework which supports not only the modification of replication factors, but also the adjustment of retry limits and deadlines, and the remapping of software components to processors.

Other work has addressed the need for a more formal study of software adaptation, particularly the reconfiguration of software modules. Bloom [8] presents an extension of Argus [42] which allows Argus "guardians" to be replaced within a software configuration. The Conic system [32] provides a two-level language, including a programming language for describing the individual software modules and a configuration language for describing the module interconnections. RESAS supports such software module reconfiguration. In addition, RESAS allows the modification of the software-to-hardware map as well as modification to
the internal structure and function of the software modules themselves. Furthermore, RESAS is designed with a particular application domain in mind — reliable real-time systems — and its programming language supports the explicit description of reliability and real-time characteristics of the software.

In the real-time systems domain, several factors affect the focus of the study of reliability.

- As noted previously, the reliability of a system includes timing considerations. A program which produces appropriate actions, but fails to meet specified timing constraints, is not reliable.

- The requirements for reliability are generally higher than in other systems. The Federal Aviation Administration guidelines for flight-critical control systems for commercial aircraft calls for failure rates of less than $10^{-9}$ per hour for a 10 hour mission, and current military aircraft guidelines are in the range $10^{-7}$ to $10^{-5}$ failures per hour [67].

- The hostile nature of the environments in which these systems are embedded may result in the reliability delivered by components of a system being no better than, and generally lower than, the reliabilities of components operating in more benign environments [64]. For example, the mean time to failure for the components of the Space Shuttle computer system during the first five flights was on the order of 6000 hours [19], and the mean time to failure for components of the digital avionics system in F-15 fighter aircraft is on the order of 4.2 hours of flight time [67]. The delivered failure rates of system components are therefore several orders of magnitude worse than those required for the system as a whole. This implies that systems must be designed to tolerate single (and possibly multiple) component failures.
Common "static" reliability techniques (e.g. triplication of all components) leads to bulky and expensive systems which nevertheless exhibit degraded reliability as their lives progress. In contrast, many biological systems have the ability to learn about and adapt to environmental conditions and their own capabilities in such a way that their "reliability" and "performance" may improve over time.

The underlying use of adaptable software in enhancing and maintaining reliability in real-time systems is in reconciling mismatches between the capabilities of the target system and the requirements of the surrounding environment. There are several reasons why adaptations may be appropriate.

- A target system may experience changing demands for its services by the surrounding environment. These changes may be due to different known phases in the mission of the system — for example, the launch, orbital and reentry phases of a space vehicle [19], or the changing demands may be due to evolutionary changes in the mission of the system — for example, changing utilization patterns in a communication switching system.

- The hardware configuration may change over time. Hardware components may fail, new hardware components may be added to replace the failed components or to increase the power of the hardware, and hardware may be reconnected in different configurations. These hardware changes may be anticipated — as in the case of component additions, or unanticipated — as in the case of component failures.

- Improved information about the characteristics of the application domain or of the software or hardware characteristics may become available. This information may arise from
testing during software development, or from data gathered during the operational life of the system.

• From a practical standpoint, it is now possible to adapt software. This is due, in particular, to the availability of sufficient hardware redundancy to allow some degree of degraded performance in the presence of hardware component faults [44, 19].

1.6 Automatic Adaptations and Real-Time Systems

In some situations, the ability to adapt automatically — without direct human intervention — is useful or necessary. There are several reasons for this:

• In some embedded systems, the target system is in a "tight" control loop within the surrounding environment. In such systems, human intervention in response to failures may be impossible within the time-frames necessary for survival.

• In other applications [8] (e.g. deep-space probes). long mission time, physical remoteness and/or hostility of the environment may make human intervention costly or impossible.

• In contrast to early reliable systems such as SIFT [77] and FTMP [14] which possessed relatively few processors and homogeneous (e.g. bus-based) interconnection structures, future reliable, embedded systems are likely to contain hundreds of processors and heterogeneous interconnection structures. These systems are too complex for anyone but "experts" to understand. Therefore, application programmers must have expert knowledge of the hardware and software characteristics or they may do more harm than good. Furthermore, even if such "experts" are available (and they may not be), the cost of supporting them may be too high in the long term.
1.7 Thesis

In light of the definitions given above and the discussions of the problem domain, the thesis of this dissertation is: A uniform software adaptation system is a viable tool for the achievement and maintenance of reliable real-time performance.

1.8 The Range of this Dissertation

This dissertation describes a general model of adaptable systems and a set of mechanisms for choosing and performing adaptations. It provides a practical outlook on the subject of adaptability as it relates to reliable real-time systems. Enough theory is presented to provide a solid framework, but emphasis is placed on the models and mechanisms needed in practical systems.

The long-term goal of this research is a system which can encompass most or all of the ad hoc adaptation algorithms and mechanisms which have been proposed for reliable real-time systems. Because the areas of adaptability, reliability and real-time systems are quite large, it is not possible to treat all of the related issues in detail. This dissertation examines a specific class of real-time system, selects a particular
class of computer hardware faults and demonstrates the usefulness of a prototype RESAS implementation using several application-independent adaptations. Adaptations are performed specifically to increase target system reliability and to ensure that the target system meets real-time deadlines. The hardware configuration is fixed, and adaptations are performed on the software components in order to most effectively employ the particular hardware configuration. Tests involve a real-time program executing on a multiprocessor consisting of eight Intel 86/30 single-board computers. Adaptation decisions are generated on an attached Sun 3/50 workstation.

It should be noted that, despite the fact that adaptations described in this dissertation are performed for a specific purpose on a particular type of computer hardware, many of the issues and mechanisms discussed in this dissertation have broader applicability. For example, if appropriate adaptation algorithms were formulated, the RESAS mechanisms could be used, with only minor changes, to enhance the availability of a bank's automated teller network.

In this dissertation, the following issues are discussed:

- What types of real-time requirements, reliability models, measures and mechanisms are appropriate for real-time systems? How may such reliability and real-time requirements and measures be formulated?

- How may the reliability and real-time characteristics of the target system be analyzed?
• What do adaptability, reliability and real-time considerations imply about the appropriate choices for the programming model, representation framework and adaptation control system?

• What mechanisms are appropriate for enacting adaptations?

• What are appropriate adaptation algorithms for the given application domain, and how may such algorithms be represented within the adaptation controller?

• How may the cost vs. benefit ratio of adaptations be analyzed?

This dissertation is related to other on-going work in the ISSOS project at the Ohio State University, including:

• How should application-dependent adaptable programs be specified and generated?

• How should such application-dependent adaptation decisions be made?

• How should information be collected on-the-fly from the target system?

• What information should be collected from the programming environment?

• How should such information be represented, stored, formatted and presented?

To the extent possible, this dissertation will make use of the other ISSOS work, pointing the reader to related papers, including: [51], [50], [62], [63], [52], [53], [54], [61], [55], [56], [57], [59], [60].
1.9 A Real-Time Software Adaptation System

The architecture of the RESAS prototype is shown in Figure 2.
The runtime system, executing on the target hardware, acts as host for the target software. It performs normal operating system functions, and monitoring (MON) and adaptation enactment (AE) mechanisms are embedded in the runtime system.

The data management system (DMS) is the repository for all state information necessary for software adaptation, including configuration information gathered during software development and monitoring information gathered during software execution. It uses a uniform entity/relationship-based (ER) representation framework. The DMS is capable of querying the runtime system for information, and of sending adaptation enactment commands to the runtime system. The ER representation framework is used to represent adaptation-related information about both the software and hardware components of the target system, and their interactions. It provides a basis for a language in which adaptations may be described.

The adaptation controller (AC), the decision making mechanism, chooses which adaptations to perform. It is the repository of the adaptation algorithms. All adaptation enactment commands pass through the DMS. The data management language (i.e. the ER representation framework) provides a uniform interface between the AC and the rest of the system. The AC and associated analysis tools may access
information in the DMS, choose adaptations, and, by modifying the representations in the DMS, cause adaptations to be enacted.

RESAS operates in two modes: the initial configuration generation (ICG) mode and the program adaptation mode.

In the initial configuration generation mode, programmers use the RESAS programming model (see Chapter 3) in conjunction with normal program generation tools (e.g. editors and compilers) to compose the program and describe valid adaptations to it. The program description is stored in the DMS, using the ER representation framework (see Chapter 4).

Using information provided by the programmer, information about software and hardware characteristics (e.g. resource requirements), and built-in algorithms (e.g. for resource allocation), the AC generates an initial target system configuration. This is described in detail in Chapter 7.

Once an initial program has been generated, it may be adapted. This may be done either statically or dynamically. The adaptations are under the control of the adaptation control system (ACS). The ACS consists of the AC, the monitoring mechanism (MON) and the adaptation enactment mechanism (AE). Thus, the ACS performs
three functions: *collection of information*, *adaptation choice* and *adaptation enactment*. Adaptation proceeds in cycles: *collect—choose—enact*. The ACS operates as a control system with feedback. Certain conditions, reported by the monitoring system and passed through analysis algorithms, trigger adaptation-choice algorithms, which pass the requested adaptations to the adaptation enactment mechanism. Static and dynamic adaptations are described in detail in Chapters 8 and 9.

1.10 Overview of the Rest of the Dissertation

The remainder of this dissertation proceeds as follows:

Chapter 2 discusses other work in real-time systems and reliability. Generally accepted terminology is presented. Real-time requirements, reliability and its characterization in typical real-time systems are discussed.

Chapter 3 discusses the requirements for a programming model for adaptable, reliable real-time software and describes the object-based model chosen for the RESAS prototype. A restriction of the object model which has been implemented is presented.
Chapter 4 discusses the requirements for a representation framework for adaptable systems and describes the entity/relationship-based framework chosen for the RESAS prototype. The mapping of target system components, such as objects and processors, to the entity/relationship framework is presented.

Chapter 5 discusses the components of the Adaptation Control System, their structures and interactions. Chapter 6 discusses the implementation of the RESAS prototype on the target hardware.

Chapter 7 discusses the generation of the initial program configuration from descriptions provided by an application programmer. Several sample adaptation algorithms are presented. Chapter 8 discusses static adaptations and presents sample static adaptation algorithms. Chapter 9 discusses dynamic adaptations and presents sample dynamic adaptation algorithms.

Chapter 10 discusses the results of several tests of the RESAS prototype, including evaluation of the performance and overhead associated with the RESAS components.

Finally, Chapter 11 discusses experience gained from the RESAS prototype and suggests areas of further research.
Chapter 2
Reliability in Real-Time Systems

This chapter gives brief overviews of the fields of real-time systems and reliability. Issues associated with the ability of adaptations to enhance the reliability of real-time systems are discussed.

2.1 Real-Time Systems

2.1.1 Definitions

A real-time system is one in which the specification of service provided by the system includes temporal information. That is, the correctness of the service provided depends not only upon the type of service but also upon timing constraints imposed on the service. A distinction may be made between different types of real-time systems [30]. A hard-real-time system is one in which violation of timing constraints
may result in catastrophic consequences. In a soft-real-time system, while timeliness is important, timing constraints may be violated as long as they are not violated too frequently or by too great a margin.

Different types of timing constraints may be specified. Dasarathy [17] classifies timing constraints into three categories:

- **Maximum**: No more than $t$ time units may elapse between two events.
- **Minimum**: No less than $t$ time units may elapse between two events.
- **Durational**: An event must occur for $t$ time units.

Dasarathy further classifies events by the nature of their participants. The *request* for service from a system is a *stimulus* and the *delivery* of the service is a *response*. There are various combinations of timing constraints. For example, *minimum stimulus-to-stimulus time* is the minimum time between two successive requests for service from a system (e.g. memory cycle time).

One of the most important timing constraints in real-time systems is *maximum stimulus-to-response time*. This is frequently characterized by a *deadline* - the time by which a system must serve a request. Algorithms given in Chapter 7 address this type of real-time constraint.

A distinction is occasionally made between *periodic* systems, which must execute with fixed periods, and *sporadic* systems which are
triggered at unpredictable times [46]. Such timing characteristics will not be discussed here. The systems will be assumed to be initiated by some external stimulus, which may be a clock (for periodic systems).

2.1.2 Typical Real-Time Systems

While there is no single "typical" embedded real-time system, many real-time systems exhibit similar features [76, 43]. Figure 3 shows a real-time system used for controlling a mechanical device (or plant). Sensory devices\(^1\) collect information about the current state of the plant and the surrounding environment. Additional sensory devices may collect commands (e.g. from a human interface) specifying a requested state for the plant. The embedded real-time system then calculates the necessary commands to send to the actuators on the plant so that the plant enters the requested state. This is essentially a feedback control system with the plant within the control cycle. The following examples illustrate these features.

In the Space Shuttle control system [19], the main control system is the flight control high frequency executive (HFE). Sensory inputs include various inertial systems, temperature and pressure sensory devices,

\(^1\)In real-time systems, input devices are frequently called "sensors". However, in this dissertation, the term "sensor" is used to denote a part of the ACS monitoring system. See Chapter 6.
Figure 3: A Typical Real-time Control System

and commands from the pilot and from the guidance system. Actuators include control surfaces, thrusters and engines. The HFE system runs at 40ms intervals and typically runs for 8ms to 12ms. Time constraints on the execution of the HFE system include a "jitter" constraint on the period of the output commands (40ms plus or minus 800μs), a "transport lag" constraint (less than 18ms between receiving sensory
device inputs and sending actuator commands) and a constraint on the number of missed HFE cycles (in the worst case, loss of 10 consecutive cycles (400ms) can result in loss of the vehicle).

In the experimental X-29 fighter [28], sensory inputs include various inertial systems, temperature and pressure sensory devices, and commands from the pilot. Actuators include control surfaces and engines. The control system in the X-29 has a 25ms control cycle.

In the Adaptive Suspension Vehicle [44], a six-legged vehicle designed to carry a human operator and payload over rough terrain, the sensory inputs include position, velocity and hydraulic actuator pressure sensory devices on the legs, an inertial reference system on the body, a laser-driven optical radar scanner and input from the human operator via joysticks, switches and a keyboard. Actuators include the hydraulic cylinders on each leg and the engine. The six leg control systems run on 10ms cycles, while the body control system runs on a 50ms cycle.
2.2 Reliability in Real-time Systems

2.2.1 Definitions

In this section, definitions of commonly used terms are given. This is only an overview - a good outline of the reliability field is given in [34], and more depth can be found in [64], [70] and [49].

As mentioned previously, a system (specifically, the target system) is a group of components providing a service to a surrounding environment. The service may cycle between periods of service accomplishment and service interruption. The service provided by a system is assumed to be specified by an authoritative system reference which is an accepted description of the expected service. This reference may not exist in a concrete sense. For example, in a triplicated set of systems providing the same service, the system reference may be approximated by majority vote, where the majority of the systems are assumed to have correctly provided the specified service, and the minority (if any) are assumed to have deviated from the specified service. The reliability of a system is the quality of service delivered by

2 The terminology in this section corresponds, as much as possible, to that suggested by the IEEE IFIP Working Group 10.4 [34].
the system such that reliance can justifiably be placed on this service. As is evident from the definition of "real-time", reliability includes meeting timing constraints. That is, a system which delivers correct services but does not meet the specified timing constraints is not operating correctly.

The study of reliability may be divided into three major topics [34]:

- **reliability impairments:** characterizations of the circumstances which may cause the service delivered by a system to deviate from the specified service,

- **reliability measures:** measures of the ability of a system to provide the specified service, and

- **reliability means:** methods of achieving and maintaining reliable service from a system.

2.2.2 Reliability Impairments

A **failure** is defined as the deviation of the service delivered by a system from the service specified. A failure is the result, visible to the surrounding environment, of errors in the system.

---

3 This is actually the definition of **dependability** [34]. However, since the term **reliability** is more common, it is used here. Reliability is also used to denote a specific measure. It is discussed in a later section.
An error (or erroneous state) is a system state which may cause a failure. An error is effective if it is currently capable of causing a failure. Otherwise, it is a latent error. An error is the result, within the system, of a fault.

A fault is defined as the underlying cause of an error. The distinction between faults and failures depends on point of view. A fault is generally considered to be the circumstance which is the "root cause" of a service deviation and the ultimate target of efforts to achieve, maintain or restore the specified service. Other service deviations which may result from a fault are considered to be failures. Thus, faults cause errors in a system, and errors cause the system to fail to deliver the specified service. The resulting failure may be considered to be a fault by some higher-level system.

Faults (and failures) may be classified by their occurrence characteristics. Permanent faults place the system into a permanent erroneous state. Temporary faults may be divided into two categories. Transient faults arise from temporary external environmental factors. Intermittent faults exist for some period of time, but may cycle between active and inactive phases.
2.2.3 Reliability Measures

In order to allow comparisons and decisions to be made about the reliability of a system it is necessary to have specific measures of the reliability of the system. Siewiorek [64] provides the following taxonomy:

- **deterministic measures**: Measures which do not depend on the probabilistic nature of faults, errors and failures. For example, K-resilient objects [7] are designed to operate correctly in spite of up to K processor node failures.

- **probabilistic functions**: Functions which represent the probability of certain situations (e.g. as a function of time). Examples include:
  - reliability: The probability that a system does not fail in the time interval \([0,t]\), given that the system was operating correctly at time 0.
  - availability: The probability that a system is operating correctly at time \(t\), given that the system was operating correctly at time 0. During the intervening time period, the system may have experienced service interruptions.
  - mission time: The time at which the reliability of a system falls below a specific level.

- **probabilistic values**: Single values which represent some characteristic of the reliability of the system. For example:
  - mean time to failure (MTTF): The average elapsed time from time 0 until a failure occurs, given that the system was operating correctly at time 0.
  - coverage: The probability that a system can recover from a specific type of failure.

The preceding concepts may be applied to many types of systems, including computer hardware and software, and mechanical systems.
The practice of embedding multiple computer systems in real-time environments has necessitated several other measures, however.

Since faults within multiple computer systems may not cause complete system failure, there is the possibility that some level of degraded performance may be available in spite of one or more failures. In such systems, the "all-or-nothing" measures (e.g. MTTF) described above do not provide the necessary descriptive power. Several performance-related reliability measures have been proposed, including computation reliability and computation availability [6] and performability [45].

As computer systems are embedded into critical real-time applications, it is not sufficient to treat all failures equally. Some failures may be benign while others may lead to catastrophic results. The concept of safety and its related measures [2, 38] provide a measurement of freedom from catastrophic failure. It can be shown that there is a tradeoff between safety and availability in some systems [2].

2.2.4 Reliability Means

The means of achieving and maintaining reliable service by fault management may be divided into two categories [34]:

- fault avoidance: Prevention of fault occurrence by properly constructing the system. These methods include using "high quality" components, providing proper shielding and grounding
of hardware components, enforcing software design rules, proper documentation and so on.

- fault tolerance: The ability to provide correct service in the presence of faults.

This dissertation is primarily concerned with fault tolerance. Therefore, the area of fault tolerance is discussed in detail. Fault tolerance may be divided into three areas [64]: fault detection, fault masking and error recovery.

2.2.4.1 Fault Detection

The underlying mechanism for fault detection is redundancy. The assumption is that, if redundant copies of a system are maintained, not all copies will be identically corrupted by faults. That is, there is some degree of fault independence between the copies. Therefore, faults may be detected by comparison of the redundant copies. Since faults may arise from any service on which the copies depend, it is important to keep their dependence on shared services to a minimum. For example, two processors which use the same power supply may both be affected by a fluctuation in the power provided by the supply. The resulting errors and faults may be identical in both copies, making comparison ineffective for detecting the power supply fault.

To guard against transient, environmentally-induced faults, the copies may be separated in space and time. Spatial separation consists
of keeping copies in different locations (e.g. keeping software copies in different memories) so that they are unlikely to be identically affected by a single local fault [70]. **Temporal** separation consists of performing actions on the copies at different times, so that transient faults are unlikely to affect all copies identically. SIFT [77] allows temporal separation by enforcing only loose software synchronization.

**Design diversity** [3, 31, 19] (also known as "N-version programming") consists of using different designs and implementations of systems providing the same service. It guards against human-induced design and implementation faults (although there is some evidence to dispute this [31]) and also against physical faults, since the effect of identical faults may be different on different implementations. For example, versions of the Space Shuttle control software were written by two separate programming teams from two different companies.

System-level fault detection mechanisms include duplication and comparison, watchdog timers and other techniques [64].

**Duplication and comparison** consists of duplicating the complete system and activating both copies in response to requests for service. The delivered services of both copies are compared by a voter.
**Watchdog timers** provide a redundant time reference against which the system may track its progress. If the timer expires, a fault is assumed to exist, because no service has been provided within the specified time.

### 2.2.4.2 Fault Masking

As with fault detection, redundancy is the primary mechanism for fault masking. Assuming that all copies of a system do not fail at the same time, and that faults are detected by some means, fault masking involves ignoring the service delivered by the faulty copies and using the service delivered by the correct copies.

**N-modular redundancy (NMR)** involves executing \( N \) copies of a system \( (N \geq 3) \) and voting on the results (usually a majority vote). NMR performs both fault detection and fault correction. Since there is usually a single voter, NMR systems are still vulnerable to single-point failures. However, the voter is typically much simpler (and therefore less likely to fail) than the \( N \) copies of the system. Many current systems use NMR, including SIFT [77], FTMP [14] and MAFT [73].

In the *Circus System* [15], a replicated procedure call mechanism may be used to allow the programmer to define *troupes* which perform services. A troupe is a set of identical systems which are logically
encapsulated within an enclosing shell. A member of one troupe may send requests for service to another troupe without knowing the replication factors of its own troupe or the other troupe. The efficiency of the troupe concept is based on the assumption that the processors are fail-stop. Failures cause a processor to stop, so that different members of a troupe do not send out different messages. The troupe concept may be made more robust (but slower) by providing a Byzantine Agreement mechanism [33] for resolving conflicting messages.

If some diagnosis mechanism is known for a system, then standby spares may be used. In this technique, a system is duplicated - there is one active copy and one spare copy. Results are compared. If there is a mismatch, diagnosis is done on both copies and, if the active copy is faulty, the active and spare designations are switched.

In reconfigurable NMR systems, if the voter detects a disagreement, the minority of the copies are replaced with spares. Typically, diagnostic and recovery mechanisms are run on the "faulty" copies and, if the copies are judged fault-free, they are placed back in the pool of spares. In continuously reconfigurable systems such as FREMP [27], spares are routinely swapped with active copies. This minimizes the effects of faulty copies and allows errors to be flushed out and found.
Adaptive voting may also be used with NMR systems. A copy which disagrees frequently with the other copies may be given less weight in the voting process. Such adaptations need not be based on the number of faults - they may be based on the criticalness of the mission. The Space Shuttle computer system requires a total of three (out of four) concurring votes for liftoff decisions but only two concurring votes for less critical decisions [19].

Rather than allowing spare systems to remain idle during normal operation, graceful degradation techniques allow all systems to provide service at all times. In the event of a system failure, the required services are shifted to the remaining systems. This technique is effective as long as the total capabilities of the remaining systems do not fall below a certain level.

The choice between active and passive (standby) redundancy depends on several factors. If the time requirements for fault masking are restrictive, active redundancy may be necessary. Conversely, in certain circumstances, idle spares may have longer lifetimes than active spares, so passive redundancy is more appropriate [64].

Because of the real-time nature of the application, fault tolerance mechanisms themselves must operate under time constraints. For
example, in the event of a computer failure in the Space Shuttle computer system, service must be restored within 400ms. Also, after a fault is detected, it may not be possible to do fault diagnosis and error recovery before computation resumes. Typically, active redundancy must be employed both to detect and mask the faults. Thus, N-modular redundancy, troupe and graceful degradation are likely candidates for use, while standby spares may not be applicable. In some cases, faults may be masked by simply ignoring the erroneous result. This is true in cases where there is some degree of redundancy built into the application domain. For example, loss of a sensory device reading may not be critical as long as some minimal number of readings are collected within some time period.

2.2.4.3 Error Recovery

Once a fault has been detected and (possibly) masked, it may still be necessary to replace a resulting erroneous system state with a correct state.

Backward error recovery consists of rolling the system state back to a previous state and continuing from that point. This previous state may have been recorded on stable storage at a previous savepoint. This technique is used frequently in database systems.
In some systems, it may not be possible to roll the system back to a previous state. A popular example is a payroll program which prints paychecks. It is difficult to "unprint" a paycheck. In such situations, forward error recovery may be required. Forward error recovery consists of calculating a new system state - one which is judged to be a valid state in light of the fault and the current state of the surrounding environment. Since, in real-time systems, the surrounding environment may not be rolled back to a previous state, backward error recovery is generally not as useful as forward error recovery.

If the service provided by the system is idempotent (i.e. two successive executions of the system has the same result as one execution), then retry may be used to recover from the error. If timing constraints allow retry to be done within the normal execution of the system, retry may be considered to provide fault masking as well.

Some reliable-programming methods integrate several constructs in an attempt to address detection, masking and recovery. Recovery blocks [21, 64], for example, combine acceptance tests for fault detection, alternate functions for fault masking and checkpointing for error recovery.
This dissertation does not address error recovery. However, it should be noted that RESAS does not interfere with the error recovery mechanisms mentioned above. In fact, programmers may make use of RESAS mechanisms, for example, to experiment with configurations containing different recovery mechanisms.

2.3 Reliability Adaptations for Real-Time Systems

The goals of a software adaptation system for the production of reliable real-time systems include:

- providing a uniform framework for implementing many or all of the reliability techniques mentioned in the preceding sections,
- allowing the application programmer to experiment with those techniques, and
- allowing the automatic selection and use of those techniques based on the programmer's suggestions and/or feedback from the target system.

In reference to the discussion in the previous sections, RESAS may be used, for example, to define and choose deadlines for the execution of software components and to set corresponding watchdog timers. Programmers may experiment with different versions of software components, including duplicated and triplicated versions, thus implementing design diversity and reconfigurable NMR. Voting weights may be
adjusted, implementing adaptive voting. Dynamic adaptations may be used to reconfigure the software in response to faults, implementing graceful degradation or standby spares; to move software from less-reliable to more-reliable processors; to adjust the number of retries allowed for software components.
Chapter 3
Programming Model

3.1 The RESAS Programming Model

The RESAS programming model provides a programming methodology which supports the production, execution and adaptation of adaptable, reliable, real-time software. It allows the programmer to create and adapt target software by providing constructs for:

- describing the functionality of the software,
- describing the reliability needs and characteristics of the software,
- describing the real-time needs and characteristics of the software, and
- describing allowable adaptations to the software.

The RESAS programming model describes software as a hierarchical collection of objects [25, 35, 1, 54]. The lowest-level objects are
called basic objects. Basic objects encapsulate data (called state) and provide operations which access and manipulate the state. State may not be shared between basic objects. All inter-object communication takes place through explicit message-passing ("by value"). All communication between objects passes through well-defined object interfaces. As a result, erroneous data cannot be passed between basic objects except within messages, and these messages may be protected from communication errors by error detecting/correcting codes.

Basic objects may be combined into composite objects. At their interfaces, basic and composite objects may be indistinguishable, although composite objects may take on forms which do not correspond to basic object forms (see Section 3.1.2).

Each object has a type, and all objects are created as instances of previously defined types\(^4\). Programmers may specify the exact type of an object instance, or they may specify a more general type and allow RESAS to choose the exact type to be instantiated.

\(^4\)The term "type" is used here to denote a "template" from which specific instances may be created. The term "object" is used here as a synonym for "object instance", unless otherwise noted.
3.1.1 Basic Objects

A basic object encapsulates sequential code and state, and is the unit of scheduling and of adaptation. Figure 4 shows a typical basic object\(^5\).

\[\text{Input Req: (I1 and I2) or (I1 and I3)}\]

\[\text{Input Interface (I1 in text)}\]

\[\text{Output Interface (Q1 in text)}\]

Figure 4: A Basic Object

\(^5\)The RESAS programming model is not a graphical programming language, although it is well suited for graphical display, and could easily make use of a graphical language such as STILE for a programmer interface.
A basic object contains a single thread of sequential code written in a sequential language (such as C). It is assumed that the execution time of this code is fixed (or has a fixed upper bound) and is known. This is a reasonable assumption in real-time systems, where the control algorithms must have bounded execution times if the plant is to be controlled correctly. In the RESAS prototype, processors are assumed to be identical, so a processor-specific measure may be used (e.g. CPU-seconds). For hardware configurations with non-identical processors, a processor-independent measure should be used, if possible, or the execution times of the object should be known for each of the processor types. In basic objects, the execution time is proportional to the amount of processing power needed. This execution time does not include the data communication times for inter-object communication. As a result of these known execution times, the RESAS adaptation controller can make decisions concerning real-time constraints, including deadline calculations and schedule generation.

A basic object may also contain modifiable state. It is assumed that this state (and, in fact, the entire object: code and state) requires a

\[6\] If objects are allowed to delay — relinquish the processor for a certain period of time — within their sequential code, then the traversal time for the sequential code may be divided more accurately into the active execution time (proportional to the needed processing power) and the delay time. For simplicity in describing RESAS, such delays are assumed to not exist here.
fixed (or bounded) and known amount of memory and that this memory is allocated statically, so that there is no inter-object interaction due to dynamic memory allocation during program execution.

Thus, resource requirements are known, which facilitates dynamic reallocation adaptations. Furthermore, the absence of shared state and of dynamic memory allocation eliminates side effects due to hidden inter-object interaction channels, facilitating reliability adaptations.

3.1.1.1 Inter-Object Communication

Basic objects communicate via "by value" messages, called invocations, containing both control and data. Basic objects pass invocations via their interfaces. **Output interfaces** of basic objects (shown as circles 1 and 2 in Figure 4 and as Q1 and Q2 in the text) may be connected to **input interfaces** of other basic objects (shown as squares 1, 2 and 3 in Figure 4 and as I1, I2 and I3 in the text) via unidirectional, asynchronous links. One-to-many, many-to-one and many-to-many interface connections are permitted. If one output interface is connected to many input interfaces, invocations appearing at the output interface are sent to all of the input interfaces (subject to return link considerations discussed in Section 3.1.2.1). Invocations arriving at an input interface are placed in an **input set** (described below) at the input interface. Basic objects may thus be connected into an "invocation
graph" describing a complex program. Furthermore, these connections may be changed dynamically, as the program is running. Multicast communication is not assumed in the RESAS programming model. Therefore, the cost of sending an invocation through an output interface is proportional to the number of links connected to the interface.

Types and roles are associated with input and output interfaces. The type of an interface describes the structure of the invocations that pass through it — “array of 80 characters”, for example. As in HPC [36], the role of an interface is a user-defined description of the purpose of invocations passing through it.

For example, an object implementing a divide operation might have two input interfaces of type integer and roles dividend and divisor respectively, and two output interfaces of type integer and roles quotient and remainder respectively. Two interfaces of an object may have the same role (e.g. an add object may have more than one addend interface). Interfaces which are connected by links must be of the same type. Type names and role names have global scope. Interface types and roles are used to check the equivalence of objects (see below). Type, and role, comparisons are by name equivalence.
3.1.1.2 The Operation of Basic Objects.

The operation of a basic object is as follows:

- Invocations arriving at each input interface are placed in the input set associated with the interface.

- When all\(^7\) input interfaces of the object have received invocations, the basic object is activated.

- From each input set, one invocation is selected for use as input data. If a particular input set has more than one invocation in it, the basic object may inspect all invocations and select one invocation. The selected invocation is deleted from the input set.

- The basic object uses the input data as input to its sequential code. The sequential code is executed using the input data and possibly the object's encapsulated state (if any). Finally, output invocations are placed in all of the basic object's output interfaces.

The sequential code within an object may send and receive invocations through the object's interfaces. The Send operation places a specified invocation in the specified output interface. The Send operation returns immediately. The invocation is not sent from the output interface, across the connected links, to other objects until the execution of the sending object is completed.

---

\(^7\)This is true during normal operation. However, in order to handle object failures, this requirement is relaxed in Section 3.4.3.
The **Accept** operation receives an invocation from the specified input interface. Since, during normal operation, an object will not execute until each input interface has at least one invocation in its input set, an invocation is returned. (Operation in the presence of faults will be discussed in Section 3.4.3.) In cases where there may be more than one invocation in an input set, the method of choosing an invocation is specified for the input interface. This input selection policy is chosen by the programmer and may be adaptable. Possible policies include FIFO, shortest-deadline-first and other, application specific, policies. For example, if an object has two input interfaces I1 and I2, it may use FIFO to choose the invocation from I1 and then choose an invocation from I2 which has the same sequence number as the invocation from I1 (or from the same link enabling set - see below).

3.1.1.3 Serializability and Concurrency Control.

The execution of a basic object is an isolated action. That is, all input invocations arrive before the execution of the object, and all outputs are presented only after the execution is completed. In general, basic objects may be executed in parallel without hidden side effects. Objects without state may always be preempted. Objects with state may be preempted by executions of other objects (since state is not shared). If the scheduling policies of the runtime system allow one
activation of an object containing state to be preempted by another activation of the same object, then it is assumed that appropriate locking mechanisms are provided and that bounds on the lock contention times are known [37] and are included as part of the object's execution time. Conversely, if the runtime system prohibits such preemption, then it is assumed that appropriate real-time non-preemptive scheduling algorithms are provided.

3.1.1.4 Input Requirements

In normal operation, a basic object uses a complete set of inputs (that is, one from each input interface). To execute correctly in the presence of erroneous or missing invocations, a basic object requires some sufficient set of correct inputs. In order to differentiate redundant parallelism (for reliability) from speed-up parallelism (for performance), an input requirements specification is provided by the programmer for each object. This specification describes the sufficient set of correct inputs needed for the object to execute correctly. For example, the specification "(I1 and I2) or (I1 and I3)" for object A in Figure 4 means that, although A normally executes when I1, I2 and I3 all contain invocations, A will execute correctly if there are error-free invocations in input interface I1 and either I2 or I3 (the "or" is an inclusive-or).
3.1.1.5 Views of Basic Objects

Figure 5 shows a syntax for the basic object type A from Figure 4 as its creator would see it.

A basic object contains three sections: an interface section, a state section and a code section. The interface section contains general information about the object and its interfaces. This information is used when interconnecting objects and is viewable from outside the object.

The state section contains static variables and (optionally) their initial values. Attributes are variables whose values are externally visible and may be viewed by the RESAS monitoring mechanism and modified by the adaptation enactment mechanism. Their values may be accessed by the adaptation controller via the data management system. LocalVariables are not visible to RESAS or to other objects.

The code section contains the single thread of sequential code for the basic object. It may also contain local dynamic variables (as in the C language). This code begins by accepting all inputs, then executes the object's application code and, finally sends all outputs.
/************************************************************************/
BeginBasicObjectType;
/************************************************************************/
BeginBasicObjectInterfaces;

ObjectIdName: A;

InputInterfaces:
  Name: I1; Type: Integer; Role: Feedback; SelectionPolicy: FIFO;
  Name: I2; Type: Integer; Role: DesiredPosition; SelectionPolicy: FIFO;
  Name: I3; Type: Integer; Role: DesiredVelocity; SelectionPolicy: FIFO;

OutputInterfaces:
  Name: O1; Type: Integer; Role: CommandedForce1;
  Name: O2; Type: Integer; Role: CommandedForce2;

InputRequirements: (I1 and I2) or (I1 and I3);

EndBasicObjectInterfaces;
/************************************************************************/
BeginBasicObjectState;

Attributes:
  Integer ExecutionTime = 100;
  Integer MemoryNeeded = 64;
  Integer NumberOfInvocations = 0;
  Float GainFactor = 1.23;

LocalVariables:
  Integer Lengths(3) = {4,7,3};

EndBasicObjectState;
/************************************************************************/
BeginBasicObjectCode;

Integer FeedbackBuffer, ReqPosBuffer, ReqVelBuffer,
         ComForce1Buffer, ComForce2Buffer;
Boolean Status1, Status2, Status3;

Accept(I1,FeedbackBuffer,Status1);
Accept(I2,ReqPosBuffer,Status2);
Accept(I3,ReqVelBuffer,Status3);

    . execute application code...

Send(Q1,ComForce1Buffer);
Send(Q2,ComForce2Buffer);

EndBasicObjectCode;
/************************************************************************/
EndBasicObjectType;
/************************************************************************/

Figure 5: A Programmer's View of a Basic Object
A basic object may be viewed in different ways, depending on the information that is of interest in a particular situation. The internal view shows all of the intrinsic attributes of the object, including the interface section, the state section and the code section. The internal view does not show the connections of the interfaces to external links, however.

The external view shows only the interface section and the attributes of the state section (which must include resource requirements (e.g. execution time).

3.1.1.6 Basic Object Equivalence

Since adaptation may involve replacing objects with other objects, it is necessary to specify under what conditions two basic objects are equivalent. There are two types of equivalence to consider. Two objects are structurally equivalent if there is a one-to-one mapping of the interfaces of one object to the interfaces of the other object such that the types and roles of the corresponding interfaces are the same. Structurally equivalent objects may replace one another in a program since they "plug into" the program in the same way. Since the functions they compute may be different, the semantics of the program may be altered, however.
Two objects are functionally equivalent if they are structurally equivalent and they compute the same function. Furthermore, the input requirement specifications and input selection policies must be identical. Functionally equivalent objects may replace one another in a program with no affect on the semantics of the program.

The structural equivalence of objects can be checked easily. The functional equivalence of objects must be specified by the programmer. The specification of functional equivalence is a topic of current research [20].

3.1.1.7 Summary

In summary, basic objects have the following properties:

- They encapsulate state and sequential code, and are the units of scheduling and of adaptation.

- They are "isolated", accepting invocations only before execution of their sequential code and sending invocations only after execution. Therefore, they may execute in parallel without unwanted side effects (subject to the possible serialization of activations of objects possessing state).

- They provide a communication mechanism which supports send-no-wait communication. They pass "by value" invocations through unidirectional, asynchronous links attached to their input and output interfaces. No other inter-object communication is possible.

- External views specify all the information necessary for connection. Objects may be replaced without worrying about inter-object side effects.
• They differentiate between redundant and speed-up parallelism by the use of input requirements specifications.

• Resource requirements and performance attributes (e.g. execution time) are fixed and known.

3.1.2 Composite Objects

In a large program, there may be many basic objects and links. Because each of these must have a unique name, the name space may grow very confusing. Also, a group of objects may provide a single service to other groups of objects, and it is useful to hide the implementation of such a service so that internal adaptations to the service will not affect the users' view of the service. For these reasons, the RESAS programming model, like many other languages [75, 36, 57], allows hierarchical composition of objects.

Composite objects are aggregations of basic (or smaller composite) objects. They have semantics which are similar to, but a slight generalization of, the semantics of basic objects. No sharing of components is allowed — there is a strict object hierarchy. An object may be enclosed in only one immediately surrounding object.

There are two distinct reasons for combining basic objects into composite objects:

• A composite object may be used as a "macro" to represent a group of basic (or composite) objects which are conveniently
manipulated as a single object. In this case, there is no need for input requirements specifications, since the composite object is not expected to have the semantics of a basic object.

- A composite object may be used as a replacement for a basic object. In this case, the semantics of the execution of the composite object, including input requirements and input selection policies, must match those of the basic object being replaced. This is the case, for example, when a basic object is replaced with a triplicated version consisting of three copies of the basic object and a voter.

Composite object types are defined only in terms of their immediate components. That is, the sub-components and sub-sub-components of a composite type are not visible in the type definition. This is because the specific types of the components, and therefore their internal structures, need not be specified, but may be chosen by the adaptation controller. The internal structure of a particular composite object instance is set when it is instantiated. Therefore, the complete component hierarchy within a composite object instance is known, and may be displayed down to the level of basic object instances.

Consider the program in Figure 8. This program was produced by taking the program in Figure 6, encapsulating objects B and C in composite object F, and encapsulating objects D and E in composite object G. Objects A, F and G were then encapsulated in object S. Object F is said to be the parent of objects B and C, and F is a child of S. (In general, however, programs are constructed one level (i.e. type definition) at a time.)
One-to-many, many-to-one and many-to-many connections are allowed between the interfaces of a parent object (e.g. F) and the interfaces of its children (e.g. B and C). Such connections may result in a communication path which appears to be a single link at one level actually consisting of many links at the lowest level. The interfaces of composite objects (and, in fact, the composite objects themselves) are not "active". That is, unlike the interfaces of basic objects, which correspond to buffer areas and code within the run-time representation of the program, composite object interfaces do not exist in the run-time representation of the program.

3.1.2.1 Call-Return Communication

As the semantics of objects have been described so far, there is provision for asynchronous, "send-no-wait" parallelism, but no provision for synchronous, "call-return" parallelism. To understand how RESAS handles synchronous parallelism, consider an analogous situation in a typical sequential language (e.g. C). A program is made up of a number of functions. When the program is linked, function calls are bound to function entries (via "call links"). At the same time, function returns are implicitly bound (via "return links") to the instructions immediately following the function calls.
Unlike call links, which may carry invocations at any time, a return link is only "enabled" when the corresponding call link is traversed. (What generally happens is that the actual address of the function entry is statically coded into the function call instruction during linking, while the return address is dynamically placed onto the stack during the function call and used by the function return instruction.)

RESAS generalizes this concept. A link may be designated as either a call or a return link. **Call links** are always enabled. When an invocation is placed on an output interface of an object, a call link attached to the interface will always transfer the invocation to the corresponding input interface attached to its other end. **Return links** are normally disabled. A return link is enabled only when every call link in its **link enabling set** has passed an invocation, and it is immediately disabled after it has passed one "result" invocation. As with input requirements specifications, link enabling sets are described by logical formulas (e.g. \(((L_1 \text{ and } L_2) \text{ or } L_3))\).
Figure 6: Call-Return Links
Consider the example in Figure 6. Suppose links L5 and L6 are return links and all other links are call links. Suppose further, that the link enabling set of L5 is \((L1)\) and the link enabling set of L6 is \((L2)\). Then object A acts as a shared server for the left- and right-hand portions of the invocation graph. An invocation passed from object B along L1 enables L5, which is used to pass a result to object C. Similarly, an invocation from object D along L2 enables L6. A’s choice of the invocation to serve enables the appropriate return link. That is, if both B and D have invoked A, A chooses one of the invocations to serve first, enables the corresponding return link, sends the return invocation and disables that return link. It then processes the other invocation.

Call links and return links need not be in a 1-1 ratio. Call links with no associated return links implement asynchronous, “send-no-wait” communication. Each return link must have at least one call link in its link enabling set, however, or it would never be enabled.

3.1.2.2 Views of Composite Objects.

Figure 7 shows a syntax for the composite object type S from Figure 8 as its creator would see it.

A composite object contains three sections: an interface section, an attribute section and a structure section. The interface section
BeginCompositeObjectType;

BeginCompositeObjectInterfaces;

ObjectTypeName: S;

InputInterfaces:
Name: I1; Type: Integer; Role: LeftEntry; SelectionPolicy: FIFO;
Name: I2; Type: Integer; Role: RightEntry; SelectionPolicy: FIFO;

OutputInterfaces:
Name: Q1; Type: Integer; Role: LeftExit;
Name: Q2; Type: Integer; Role: RightExit;

InputRequirements: NONE;

EndCompositeObjectInterfaces;

BeginCompositeObjectAttributes;

Integer ExecutionTime = A.ExecutionTime +
MAX( F.ExecutionTime + L3.InvocationTime + L5.InvocationTime,
G.ExecutionTime + L4.InvocationTime + L6.InvocationTime );

Integer MemoryNeeded = F.MemoryNeeded + G.MemoryNeeded +
A.MemoryNeeded;

Integer NumberOfInvocations = A.NumberOfInvocations;

EndCompositeObjectAttributes;

BeginCompositeObjectStructure;

ChildObjects:
Name: F; Type: TwoPassAlgoX;
Name: G; Type: TwoPassAlgoY;
Name: A; Type: CommonAlgo;

ChildToEndLinks:
Name: L3; Invoker: F; OutInterface: Q1; Invokee: A; Ininterface: I1;
Name: L4; Invoker: G; OutInterface: Q1; Invokee: A; Ininterface: I1;
Name: L5; Invoker: A; OutInterface: Q1; Invokee: F; Ininterface: I2;
Name: L6; Invoker: A; OutInterface: Q1; Invokee: G; Ininterface: I2;

LinkEnablingSets:
Name: L5; EnablingSet: { L3 };
Name: L6; EnablingSet: { L4 };

ParentToEndLinks:
Name: L1; ParentInterface: I1; Child: F; ChildInterface: I1;
Name: L2; ParentInterface: I2; Child: G; ChildInterface: I1;
Name: L7; ParentInterface: Q1; Child: F; ChildInterface: Q2;
Name: L8; ParentInterface: Q2; Child: G; ChildInterface: Q2;

EndCompositeObjectStructure;

EndCompositeObjectType;

Figure 7: A Programmer's View of a Composite Object
contains general information about the object and its interfaces. This information is used when interconnecting objects and is viewable from outside the object.

The attribute section contains attributes, which, unlike those of basic objects, do not correspond to actual variables in the program. Their values are functions of the structure of the composite object and of the values of attributes of the child objects and links. Attributes may be accessed by the adaptation controller via the data management system, which computes their values.

The structure section describes the internal components of the composite object, including child objects and the links interconnecting them with one another and with the parent object.

As with basic objects, composite objects may be viewed in several ways. The internal view shows all of the intrinsic attributes of the object, including the interface section, the attribute section and the structure section. The internal view does not show the connections of the defined object’s interfaces to external links, however. The external view shows only the interface section and the attribute section.
3.1.2.3 Naming Components of Composite Objects

The global names of the components (child objects, interfaces and links) of a parent object are formed by appending their local names to the global name of the parent. For example, the global name of object A is S/A, while the global name of C is S/F/C. It is therefore acceptable to have three links with local name L1, since their global names are S/L1, S/F/L1 and S/G/L1. Similarly, all of the objects use interface name II, but their global names are S/I1, S/F/I1, S/F/B/I1, and so on. Because the scope of the local name of a component object is limited to the interior of their parent, it would be acceptable, for example, to change E’s local name to “C”, since S/F/C could not be confused with S/G/C.

3.1.2.4 Composite Object Equivalence

As with basic objects, composite objects are structurally equivalent if their interfaces match. Composite objects are functionally equivalent if they are structurally equivalent and they compute the same function. However, they need not have identical input requirement specifications. (In fact, they need not have any input requirement specifications.) Since composite objects may invoke other objects in mid-execution (as F invokes A in Figure 8), these intermediate outputs must be identical, and given identical invocations on the corresponding “return” input interfaces
Figure 8: Composite Objects and Names
(if any), the composite objects must compute the same function from that point on. Therefore, functionally equivalent composite objects may replace one another in a program without changing the semantics of the program. Functional equivalence is specified by the programmer using type hierarchies described in Section 3.1.3.

3.1.2.5 Composite Object - Basic Object Equivalence

Since it is useful in some situations to replace a composite object with a basic object, or vice versa, it is necessary to specify the conditions under which a composite object is equivalent to a basic object. A composite object is structurally equivalent to a basic object if their interfaces match. A composite object is functionally equivalent to a basic object if they are structurally equivalent, compute the same function, and if their input requirements and input selection policies are identical. Functionally equivalent objects are perceived identically from within the system.

In general, the input requirement specification and input selection policies of a composite object must be specified by the programmer. However, when a single child basic object is connected (via one-to-one interface connections) to all of the input interfaces of its parent composite object, the composite object inherits the input requirement specification of the child object.
Two composite objects which are functionally equivalent to the same basic object are said to be basically functionally equivalent. This is stronger than functional equivalence since it implies that both composite objects have the same input requirements specification and input selection policies.

3.1.3 Concrete and Synthetic Types

Objects are created as instances of previously created types. Types which correspond to actual object instances are called concrete types. It is also possible to create synthetic types. Synthetic types do not correspond to actual object instances. They are a way of conceptually grouping concrete types into sets with common characteristics. Synthetic types are used to create type hierarchies. A type may be a subtype of zero, one or more other types. All "non-leaf nodes" in a type hierarchy are synthetic types. All "leaf nodes" in a (complete) type hierarchy are concrete types.

Synthetic types are particularly useful for describing groups of concrete types which are functionally equivalent (i.e. provide the same services). Figure 9 shows part of a type hierarchy describing object types solving simultaneous linear equations (i.e. Ax=b). The root type "Simultaneous Linear Equation Solution Type" is a synthetic type which users may specify when any solution method is acceptable. If more is
known about the matrices involved (e.g. they are large, sparse matrices), a more informed choice may be specified by requesting, for example, an "Iterative Method Type" object. If still more is known about the matrices involved (e.g. they are strictly diagonally dominant), a specific concrete type, say "Gauss-Seidel Method Type" may be specified.

Figure 9: A Type Hierarchy
3.2 Discussion

The choice of programming model depends on the program attributes which are of interest in a particular application area. In RESAS, these attributes include: *adaptability, parallelism, reliability* and *timeliness*. This section examines these attributes and the effect these attributes have on the RESAS object model.

3.2.1 Adaptability Considerations

Description of allowable adaptations and of decision algorithms hinges on two properties of a programming model:

1. A “semantically strong” model describes the interactions of software components at a level which makes allowable and appropriate adaptations evident.

2. A “descriptively strong” model gives the programmer the power to describe the adaptations and algorithms.

(1) and (2) are, in a sense, complementary properties. That is, a model that provides strong mechanisms for users to describe adaptations may be able to compensate for a semantically “weaker” model, since the programmers may be able to make use of application-dependent properties of the programs when making adaptation decisions. Conversely, a semantically strong model may make some types of adaptations application-independent, allowing them to be specified once for all
applications. The RESAS programming model is both semantically and descriptively strong within the domain of real-time performance and reliability. The well-defined operation of basic objects and their well-defined encapsulation in composite objects allows their behaviors and needs to be analyzed with application-independent algorithms. Furthermore, the synthetic type concept allows programmers to state allowable object type choices for particular components of a program.

The RESAS programming model supports the production of software with known resource requirements and known performance characteristics. This facilitates the development of adaptation algorithms, since such information is explicitly visible and may be used by the algorithms.

3.2.2 Parallelism Considerations

The "send-no-wait" and "call-return" communication mechanisms provided by the RESAS programming model support both synchronized and unsynchronized parallelism, both of which are useful in parallel programming [80, 81, 72]. The asynchronous invocation construct distributes control as a default, with explicit synchronization only where required by the application. Since basic objects cannot share state, object executions may, in general, preempt one another. As a result, parallelism is not artificially restricted.
3.2.3 Reliability Considerations

The input requirements specifications provided by the RESAS programming model supports the distinction between redundant parallelism introduced for reliability and speed-up parallelism introduced for performance. It allows the specification of the services provided by objects (via object types), and allows the description of failures with respect to these services. The modular nature of objects, absence of shared state, invocation "by-value" and the well-defined interfaces between objects facilitates fault containment [49, 69].

3.2.4 Real-time Considerations

The RESAS programming model supports the production of software with known timing requirements and known timing characteristics.

The single thread of sequential code within a basic object and the well defined aggregation of basic objects into composite objects allows timing characteristics of basic and composite objects to be calculated. The explicit representation of asynchronous "send-no-wait" communication facilitates the analysis of communication times. The separation of object execution times from invocation times facilitates the construction of adaptation algorithms for changing the real-time performance of the target software.
3.3 Related Research

Work in adaptable systems has generally been in the area of reconfigurable systems [29]. Work in reconfigurable software has generally presupposed a multi-level software model, based on low-level building-blocks which may be connected in various ways to form different configurations. These building-blocks and their connections are the units of adaptation.

Argus [42] encapsulates resources within guardians, and guarantees that specified state variables are accessed atomically, using remote procedure calls for inter-guardian communication. In Bloom's extension of Argus '81, the guardians are the adaptation building-blocks. They may be replaced or reconnected within a configuration.

Kramer [32] describes the two-level CONIC language, which includes (1) a programming language (e.g. Pascal extended with communication primitives) for writing individual software components, and (2) a configuration language for describing the logical structure of a system composed of a set of components.

In contrast to those systems, RESAS allows not only reconnection of objects and remapping of objects to hardware, but also modifications
to internal attributes of the objects themselves. For example, if an object implements an idempotent function of its inputs, it is possible to modify the maximum number of attempts to be made (i.e. the maximum number of retries allowed) before admitting defeat and giving up.

The ISSOS project [52], [53], [54], [61], [55], [56], [57], [59], [60], allows the programmer to choose the units of adaptation, which may range in size from individual variables to groups of modules, and to specify the allowable adaptations to these units. RESAS is essentially a restriction of ISSOS, concentrating on the domain of reliable real-time systems.

Srini [68] proposes a model of adaptable software which uses extended dataflow graphs. Real-time and reliability analysis may be done on the dataflow graphs. Such graphs are easily adaptable. Nodes do not produce side effects and may be replaced by equivalent nodes without concern for state consistency. Unfortunately, since the model does not support the concept of state, it is impossible to program arbitrary real-time applications (without passing the state around in all messages). The RESAS object model allows dataflow analysis for error propagation, but it also allows the explicit representation of state.
HPC [36] allows the hierarchical composition of objects (and provides mechanisms for explicitly sharing objects by subverting the hierarchy). Communication is by unidirectional asynchronous messages (although synchronous message mechanisms may be built from them). Programmers may write application-specific adaptation algorithms for component objects within the hierarchy. The HPC and RESAS programming models have some similarities (e.g. interface specifications) but HPC is not "semantically strong" in the domain of reliable real-time systems.

STILE 75', a graphical programming language for real-time software, allows the hierarchical composition of "boxes" and provides low-level control and data communication primitives from which higher-level operations can be constructed. It does not provide any reliability or real-time specification constructs. It has been suggested, however, that the RESAS object model could be built on top of STILE with little trouble.

The Circus System [15] facilitates dynamic reliability adaptations by allowing multiple copies of a software component to be encapsulated within a troupe. Users of the troupe have no knowledge of the number of copies within the troupe. Therefore, copies may be dynamically added to, and deleted from, the troupe in response to varying performance and
reliability needs. Unlike RESAS, Circus supports only replication as a reliability mechanism and does not address real-time requirements.

NIL [72] provides processes with non-sharable state, which may be passed from one process to another, but which may only be owned by one process at a time. Both synchronous and asynchronous communication are supported. Like RESAS interfaces, NIL "ports" may be reconnected dynamically, allowing program reconfiguration. Unlike NIL, RESAS actively supports the notion of "input requirements" which allows the description of module redundancy.

Mok [46] uses processes (which correspond roughly to RESAS composite objects) as programming units and breaks them down into scheduling blocks (which correspond to RESAS basic objects) for processor assignment. Module redundancy is not supported.

3.4 A Restriction of the Object Model

In the previous sections, a general object model for adaptable, reliable real-time software was proposed. This model is quite general and allows some important parameters of the software to be chosen by the application programmer (e.g. input selection policies). The generation of
adaptation algorithms which depend on such application-specific parameters is beyond the scope of this dissertation. Therefore, in this section, a restriction of the model is described and practical implementation issues are discussed. This restricted model has been implemented and is the basis for the algorithms and mechanisms described in the remainder of this dissertation.

3.4.1 Input Interfaces

In the restricted model only one-to-one connections are allowed between the interfaces of the parent object and the interfaces of its children. This is to facilitate analysis of communication overheads. Since multicast communication is not assumed, many-to-one or one-to-many connections could result in a connection which appears to be a single link at one level actually consisting of many links at the lowest level. The one-to-one criterion ensures that one link carries one invocation per object execution. It is possible, of course, to insert explicit "copy" objects which receive a single invocation and send copies of the invocation to multiple objects.

In the general object model, each object may select input invocations from its input interfaces in any "fair" way. These may include application-independent policies, such as FIFO and random selection, as well as policies which depend on the contents of the invocations.
themselves, including shortest-deadline-first, and giving priority to invocations arriving from specific objects or across specific links. Complex call-return interconnections with large, intersecting link enabling sets may be created. While this flexibility provides the programmer with a powerful model, software written in this general model may be difficult to analyze. Since such analysis is one of the goals of this dissertation, a particular input selection policy which is more amenable to analysis is used. It works as follows:

Objects with a single input interface and a single output interface use a shortest-deadline-first input selection policy (deadlines will be discussed later). Objects with more than one input or output interface have only a single buffer in each input set. Multiple links may be connected to such an input interface, but it is the responsibility of the application programmer to ensure that the software is written so that invocations to the interface are mutually exclusive. That is, it must not be possible for a second invocation to be sent to the interface until the first invocation has been processed. This implies that only objects with single input and output interfaces may be used as shared servers which are asynchronously invoked by a number of other objects.
3.4.2 Real-Time Programs

It is assumed that real-time programs have the following form: The program is entirely encapsulated within an object $S$ which represents the entire software system. A single (conceptual) external START object is connected to all of the input interfaces of $S$. Similarly, one or more (conceptual) END objects ($END_1, END_2,...$) are connected to some or all of the output interfaces of $S$. (A single END object may be connected to more than one output interface of $S$, and its input requirements are specified exactly as those of actual basic objects.) See Figure 16.

Program execution consists of repeated executions of $S$, either periodically or sporadically. One execution of $S$ consists of an invocation from START to all input interfaces of $S$, followed by the execution of $S$, followed by invocations from the output interfaces of $S$ to the END objects. Invocations generated by START contain the BaseStartTime of the invocation, and this time is passed through $S$ within each invocation generated by $S$. This BaseStartTime is a unique identifier of the particular execution of $S$.

Each END object is provided, by the programmer, with a specified deadline, which is the elapsed time, relative to the current BaseStartT ime, by which that particular END object must be invoked by $S$. In Chapter 7 it will be shown how this information is used to choose deadlines for the basic objects which make up $S$. 
3.4.3 Object Scheduling

The general model proposes that an object be scheduled for execution whenever all of its input interfaces have received invocations. While this is acceptable when all objects are performing properly, it nullifies the power of the redundant input interfaces (specified via the input requirements specification) in situations where no invocation is sent to a particular interface (due to an "upstream" failed object).

Therefore, it is assumed that an object uses the BaseStartTime within an invocation to compute the latest time by which it must start in order to complete by its deadline. If some of the other input interfaces have not received invocations by that time, but sufficient input interfaces with respect to the input requirements specification have been invoked, the object begins execution anyway. Any Accept operations on those interfaces return failure status, so that the application code can take appropriate action. This assumes that an object receives a valid invocation on at least one of its input interfaces. This is a reasonable assumption, since if no good invocation is received, it is unlikely that the object can perform any appropriate actions.
3.5 An Example

Consider the control program shown in Figure 1. Object execution times and invocation message sizes (in bytes) are shown. Each object in the figure requires all input interfaces to be invoked before the object begins execution. Therefore, the input requirements specifications are of the form "I1 and I2 and I3 and...".

This program may be encapsulated in a System object type, with one System input interface attached to the input interface of the DataAcquisition object and one System output interface attached to the output interface of the OutputControl object, as in Figure 10. (Interfaces of the other objects are not shown in the figure.)

A conceptual START object may be attached to the System input interface and a conceptual END object may be attached to the System output interface. The END object may be assigned an overall deadline, which is the maximum allowable elapsed time from the invocation to the System input interface until the invocation from the System output interface.

Some of the objects (e.g. InverseJacobian) may implement idempotent functions and therefore it may be allowable to retry the function
Figure 10: Control Program
if it fails on the first attempt. This is controlled by an internal attribute, MaxTries, of the object.

It is not necessary that the objects be basic. In some cases (e.g. ForceTransformation), it may be acceptable to provide a basic type and a composite, duplicated type, as shown in Figure 11.

![Diagram](image)

**Figure 11:** Duplicated Force Transformation Type
The corresponding type hierarchy is shown in Figure 12. The description of the System type specifies AnyForceTransformation as the type for that child object. The adaptation controller is then allowed to make the decision between ForceTransformation and DupForceTransformation.

![Diagram of Force Transformation Type Hierarchy]

**Figure 12:** Force Transformation Type Hierarchy

Object instantiation will be covered further in Chapter 7.
Chapter 4
Representation Framework

4.1 Introduction

In order to effectively use the object model described in the previous chapter, it is necessary to represent the components of the target system. These components include basic and composite object types and instances, object interfaces and attributes, synthetic object types, processors and hardware communication channels. In particular, the adaptability, reliability and real-time characteristics of the components must be represented. This representation may then be used during the creation and manipulation of the target software, as a framework for the RESAS data management system (DMS), and as a uniform description of the target system which can be used by all of the other RESAS components (e.g. AC, MON, AE).
4.2 The Augmented Entity/Relationship Model

4.2.1 Entities and Relationships

The representation framework chosen for the RESAS prototype is an entity/relationship model [12, 47] augmented with action routines [23, 79]. An entity is a collection of attributes describing a particular component or group of components. These attributes possess values which may be fixed for the lifetime of the entity, or they may vary. In theory, any hardware or software component can be represented as an entity. In practice, the granularity of components is fixed at some size. Components of smaller size are considered to be internal attributes of entities. For example:

```
ENTITY
Name: MotionPlanning
Type?Inst: TYPE
TypeLevel: CONCRETE
InstLevel: BASIC
InputRequirements: (II)
ExecTime: 29ms
MemNeeded: 8K
MaxTries: 1
```

An entity contains no information about how components interact. Such information is represented by relationships. Relationships contain attributes describing the relationship itself and the entities that participate in it. For example:
RELATIONSHIP
Name: PROCESSOR-ASSIGNMENT
Tuples:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>S/DA</td>
</tr>
<tr>
<td>P2</td>
<td>S/MP</td>
</tr>
</tbody>
</table>

As can be seen, a relationship contains a table of interaction information (much like a relation in a relational model), and possibly additional information about the interaction itself (e.g. the name attribute).

In the entity-relationship model, adaptations are simply changes to the entities and relationships. Entities and relationships may be manipulated by operations that are similar to those used in relational database systems [47] (e.g. project, select, join).

The entity-relationship model possesses the descriptive power to represent the necessary information about the target system. Furthermore, it allows the attributes of a component which are intrinsic to that component to be separated from attributes which depend on how the component interacts with other components.

Operations on the entities and relationships allow the compilation of specific views of the system. Since adaptation frequently involves modification of particular components and component interactions, this aids the decision-making process by hiding aspects of the system which
are irrelevant to a particular decision, while making relevant aspects explicit.

**Action routines**, much like those used in syntax-directed editors [23] and frame systems [79] may be associated with each attribute of any entity or relationship. Accessing (i.e. reading or modifying) an attribute automatically triggers the execution of the corresponding action routine.

The action routines complement the *declarative* entity/relationship structure by providing a mechanism for describing relationships which are most easily expressed *procedurally*. Probably the most important use of action routines is to automatically pass modifications of the target system representation stored in the RESAS data management system to the appropriate adaptation enactment mechanisms in the RESAS runtime system.

Action routines may be used to remove "hidden" side-effects of adaptations by allowing the effect of a modification of one entity or relationship on other entities and relationships to be described within the framework. They may also be used, for example, to describe the attributes of a composite object in terms of its child basic objects. For example, the memory needed by a composite object is the sum of the memory needs of its child objects.
4.2.2 The Format of Entities and Relationships

The basic unit of information in the prototype RESAS entity/relationship framework is the attribute-value pair. For example:

\[(\text{exectime} \ 100\text{ms})\]

The set of admissible values for each attribute is called the domain of the attribute. Domains may include integer and floating point numbers, character strings, attribute-value pairs, names of entities and relationships, and sets of the aforementioned. The value \textit{nil} corresponds to the empty set.

Attribute-value pairs may be combined into records. For example:

\[
((\text{name ObjectX})(\text{exectime} \ 100\text{ms})(\text{memneed} \ 50\text{K}))
\]

Two records are said to match if, for each attribute that appears in both records, the corresponding values are equal. Attribute names which do not appear in both records are ignored. Two integers or floating point numbers match if they are equal. Two character strings match if they are equal.

An entity is simply a record which contains information intrinsic to a particular component or set of components. In particular, it does not contain any attributes whose values are names of other entities or
relationships. However, the notion of entity is stronger than such syntactic characteristics suggest. It is assumed that the programmer will not create entities whose attribute values depend on any external entity or relationship (except in the cases where action routines are used to describe "procedural" relationships). Every entity has a name attribute. The name of an entity must be unique and may be used to refer to the entity. A typical entity might look like the record shown in the preceding paragraph.

A relationship is a record with the same characteristics as an entity, except that it has one special attribute, called tuples, whose value is a set of records. Each of these records is called a tuple and its attributes are called fields. Every tuple in a particular relationship has the same set of field names. The value of a field may be the name of an entity or relationship. For example:

```plaintext
((Name PROCESSOR-ASSIGNMENT)
 (Tuples
  (((Object ObjX)(Processor P1))
   ((Object ObjY)(Processor P1))
   ((Object ObjZ)(Processor P2))
 ))
)
)
```

This relationship contains two attributes: name and tuples. As with entities, the name attribute is required and must be unique. The
*tuples* attribute contains three tuples. Each tuple has two fields: Object and Processor. ObjX, ObjY and ObjZ are the names of entities representing three software objects. P1 and P2 are the names of entities representing two processors.

4.2.3 A Note on Syntax

The RESAS DMS prototype was implemented in Lisp. However, to improve readability, the syntax used in the examples has been "cleaned up" somewhat. Entities and relationships will be shown as "tables" without Lisp's parentheses. Functions will use a "C-like" syntax, and the Lisp "quote" function will be shown as double quotes.

In all of the examples in the remainder of this work, the following contractions will be used: *Ent* for entity names, *Rel* for relationship names, *Att* for attribute names, *Val* for attribute values, *Tup* for relationship tuple names, and *Field* for tuple field names.

4.2.4 Operations on Entities

This section describes the operations which may be used to create, query and modify entities and relationships. Readers interested in an overview of RESAS may quickly skim this section in order to get a basic idea of the types of operations available.
4.2.4.1 Creation

The `Ecreate` operation creates an entity. Its format is:

```c
Ecreate( Ent, Att1, Val1, Att2, Val2, ... );
```

This operation creates an entity named `Ent` (i.e. the variable `Ent` will "point to" it) with attribute names `Att1`, `Att2`, and so on, and corresponding attribute values `Val1`, `Val2`, and so on. All attribute names must be specified in the `Ecreate` call. That is, it is not possible to add extra attributes after an entity is created. An entity must have a `name` attribute. This is used to access the entity, and the name may appear in the fields of a relationship. An initial value must be given to each attribute. This value may be "?", which signifies an unknown value.

4.2.4.2 Attribute Manipulation

Attribute values may be read and modified with the `GetAtt` and `PutAtt` operations:

```c
Val := GetAtt( Ent, Att);

PutAtt( Ent, Att, Val);
```

If the specified entity or attribute does not exist, an error is reported.
An attribute value which is a set may be read and written as a unit with GetAtt and PutAtt. In addition, new elements may be inserted and deleted with the InsertAtt and DeleteAtt operations respectively. Their formats are:

\[
\text{InsertAtt( Ent, Att, NewElement );}
\]

\[
\text{DeleteAtt( Ent, Att, Element );}
\]

InsertAtt will insert the NewElement only if it does not match any element of the set. Set values are provided primarily so that relationships may be represented in the same way as entities. In general, most "sets" are more appropriately represented as relationships. For example, the set of all basic object instances assigned to a particular processor is better represented via the PROCESSOR-ASSIGNMENT relationship shown above.

4.2.5 Operations on Relationships

4.2.5.1 Creation

The \textbf{Rcreate} operation creates a relationship. Its format is:

\[
\text{Rcreate( Rel, Att1, Val1, Att2, Val2, ... );}
\]

This operation creates a relationship named Rel with attribute names
Att1, Att2, and so on, and corresponding attribute values Val1, Val2, and so on. All attribute names must be specified in the Rcreate call. That is, it is not possible to add extra attributes after a relationship is created. If the relationship is to be initially empty, the value of the tuples attribute should be nil. A relationship must have a name attribute. This is used to access the relationship, and the name may appear in the fields of a relationship.

4.2.5.2 Attribute Manipulation

As with entities, the attribute values of relationships may be read and modified with the GetAtt and PutAtt operations:

Val := GetAtt( Rel , Att );

PutAtt( Rel , Att , Val );

If the specified entity or attribute does not exist, an error is reported. PutAtt does not check the domain of the new value. A relationship may be “emptied” by:

PutAtt( Rel , tuples , nil );

As with entities, InsertAtt and DeleteAtt may be used to insert and delete elements of a set attribute value.
4.2.5.3 *Tuples* Attribute Manipulation

The *tuples* attribute of a relationship is similar to a relation in a relational database system. Operations similar to relational operations are provided for manipulation of the relationship tuples attribute. However, unlike the standard relational model, tuples are not "owned" by a single "base" relationship. A tuple may be a member of multiple relationships, and may be inserted into, or deleted from, one relationship without affecting the other relationships. The RESAS ER model is therefore "view" oriented. New relationships are created from old relationships by copying "pointers" and not the actual tuples (unless done explicitly via a Copy operation).

The **Runion** operation creates a new relationship Rel3 with a *tuples* attribute which is the *relational union* of the *tuples* attributes of the given relationships. The format is:

\[ \text{Rel3} := \text{Runion}( \text{Rel1} , \text{Rel2} ); \]

The new relationship Rel3 will contain all of the tuples of Rel1 as well as those tuples of Rel2 which do not match any tuples of Rel1. The individual tuples themselves are not copied. That is, the same physical tuple will reside in both Rel1 and Rel3, or Rel2 and Rel3 (or all three, if Rel1 and Rel2 had the same origins!). Therefore, if the
value of a field is modified in a tuple, the change will appear in all applicable relationships. The other (non-tuples) attributes of Rel3 will be copies of the corresponding attributes of Rel1.

The Rintersect operation creates a new relationship Rel3 with a tuples attribute which is the relational intersection of the tuples attributes of the given relationships. The format is:

\[ Rel3 := \text{Rintersect}(\text{Rel1}, \text{Rel2}) ; \]

The new relationship Rel3 will contain those tuples of Rel1 which match tuples of Rel2. The other (non-tuples) attributes of Rel3 will be copies of the corresponding attributes of Rel1.

The Rdifference operation creates a new relationship Rel3 with a tuples attribute which is the relational difference of the tuples attributes of the given relationships. The format is:

\[ Rel3 := \text{Rdifference}(\text{Rel1}, \text{Rel2}) ; \]

The new relationship Rel3 will contain those tuples of Rel1 which do not match tuples in Rel2. The other (non-tuples) attributes of Rel3 will be copies of the corresponding attributes of Rel1.

The Rselect operation creates a new relationship Rel2 with a tuples attribute which is the relational selection of the tuples attribute of the given relationship. The Rselect operation has the following form:
\texttt{Rel2 := Rselect( Rell , Selector1 , Selector2 , ... );}

This operation produces a new relationship with all tuples of \texttt{Rell} whose fields match all of the \texttt{selectors}. The format of a selector is:

\begin{verbatim}
( FieldName ComparisonFunction Comparand )
\end{verbatim}

Comparison functions include: =, ≠, >, <, ≥, ≤, ∈, and ∉. For example:

\begin{verbatim}
( Size > 100 )
( Name = "Tom" )
( Day ∈ { "Monday", "Wednesday" } )
\end{verbatim}

The comparand may be constant or variable. For example:

\texttt{VarX := GetAtt(...)}

\texttt{Rel2 := Rselect( Rell ,}
\begin{verbatim}
( name = "Tom" ) ,
( time > VarX ) ,
( size < GetAtt(...) ));
\end{verbatim}

As with the previous operations, changes to individual tuples in \texttt{Rel1} will also appear in \texttt{Rel2}. Other non-\texttt{tuples} attributes of \texttt{Rel2} will be \textit{copies} of the corresponding attributes of \texttt{Rel1}.

The \texttt{Rproject} operation creates a new relationship \texttt{Rel2} with a \texttt{tuples} attribute which is the \textit{relational projection} of the \texttt{tuples} attribute.
of the given relationship. New field names may be given to the fields in Rel2. The format is:

\[ \text{Rel2 := Rproject( Rel1, FieldName1, FieldName2, \ldots, NewName1, NewName2, \ldots );} \]

The other (non-tuples) attributes of Rel2 will be copies of the corresponding attributes of Rel1.

Several operations return statistics of the tuples of a relationship. The \text{Rcount} operation returns the number of tuples in the relationship. Its format is:

\[ \text{Val := Rcount( Rel );} \]

The \text{Rsum} operation returns the sum, over all tuples in the relationship, of the values (which must be numerical) of the named field. Its format is:

\[ \text{Val := Rsum( Rel, Att );} \]

4.2.5.4 The Sort Operation

Although the tuples of a relationship are not generally considered to be in a particular order, it is possible to sort them based on the numerical values of a particular field. This is helpful when accessing them in a particular order improves an algorithm's performance and during the display of a relationship to a programmer. The format is:
Rsort( Rel , Field , ComparisonFunction );

The ComparisonFunction is ≤ or ≥.

4.2.5.5 The Copy Operation

As mentioned above, new relationships are created from old relationships by using the original tuples and creating a new relationship with pointers to those tuples. While this leads to easy and efficient compilation of views, it may also lead to such views becoming incorrect at some later time, since tuples within a view may be modified by operations on some other relationship which shares some of those tuples. Such side effects may be prohibited by making a new copy of the relationship, with new tuples. A copy of any entity, relationship, tuple or field may be created via the Copy operation. The format is:

NewName := Copy( OldName );

The new copy contains identical information, but changes to one copy do not affect any other copies.
4.2.5.6 Accessing Individual Tuples

Several operations are provided for selecting individual tuples of a relationship. The \texttt{Rmax} and \texttt{Rmin} operations return a tuple with the maximum (numerical) value in the specified field. Their formats are:

\begin{verbatim}
Tup := Rmax( Rel , Field );
Tup := Rmin( Rel , Field );
\end{verbatim}

The \texttt{Rfirst} operation returns the first tuple in a relationship. No specific tuple order is kept in relationships, but this operation is useful for accessing a tuple in a relationship containing only one tuple, and when any tuple is acceptable. The format is:

\begin{verbatim}
Tup := Rfirst( Rel );
\end{verbatim}

For accessing all tuples in a relationship, the \texttt{ForTuple} iterator is provided. Its format is:

\begin{verbatim}
ForTuple Tup In Rel Do
{
  ...
  ...
}
\end{verbatim}

For selecting a specific tuple in a relationship, the \texttt{Rselect1} operation is provided. The format is:

\begin{verbatim}
Tup := Rselect1( Rel , Selector1 , Selector2 ,... );
\end{verbatim}
Rselect1(...) is equivalent to Rfirst( Rselect(...) ), but much faster, since it does not create a new relationship and it returns the first matching tuple.

4.2.5.7 Tuple Manipulation

Tuples may be created by the Tcreate operation. Its format is:

Tcreate( Tup , Field1 , Val1 , Field2 , Val2 ,.. );

As with entities and relationships, all fields must be specified at tuple creation time.

The values of the fields of a tuple may be read and modified in exactly the same way attributes of an entity are accessed:

Val := GetAtt( Tup , Field );

PutAtt( Tup , Field , Val );

The Rinset operation inserts tuples into a relationship. The tuple will be inserted only if it does not match any tuple already in the relationship. The format is:

Rinsert( Rel , Tup );

which is equivalent to:

InsertAtt( GetAtt( Rel , tuples ) , Tup );
This action inserts tuples only into the specified relationship. If several relationships share other tuples in the relationship, it will not affect the other relationships.

The **Rdelete** operation deletes tuples from a relationship. All tuples in the relationship which match the specified tuple will be deleted from the relationship. The format is:

\[ \text{Rdelete( Rel, Tup )}; \]

which is equivalent to:

\[ \text{DeleteAtt( GetAtt( Rel, tuples ) , Tup )}; \]

This action deletes tuples only from the specified relationship. If several relationships share a tuple, it will not affect the other relationships.

4.2.6 Operations Combining Entities and Relationships

The entity-relationship equijoin **RjoinER** has the following format:

\[ \text{Rel2 := RjoinER( Rel1, Fld )}; \]

This operation uses the value of the specified field in each tuple as the name of an entity and joins the attributes of the entity onto the fields of the tuple. Redundant attributes are removed from the tuples. For example, for the following relationship and entities,
ENTITY
Name: P1
MemSize: 512K

ENTITY
Name: P2
MemSize: 256K

RELATIONSHIP
Name: PROCESSOR-ASSIGNMENT
Tuples:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>S/DA</td>
</tr>
<tr>
<td>P1</td>
<td>S/MP</td>
</tr>
<tr>
<td>P2</td>
<td>S/IJ</td>
</tr>
</tbody>
</table>

the RjoinER operation

ASSIGN-SIZE :=
RjoinER( PROCESSOR-ASSIGNMENT , Processor );

would produce the following relationship:

RELATIONSHIP
Name: ASSIGN-SIZE
Tuples:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Object</th>
<th>Name</th>
<th>MemSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>S/DA</td>
<td>P1</td>
<td>512K</td>
</tr>
<tr>
<td>P1</td>
<td>S/MP</td>
<td>P1</td>
<td>512K</td>
</tr>
<tr>
<td>P2</td>
<td>S/IJ</td>
<td>P2</td>
<td>256K</td>
</tr>
</tbody>
</table>
4.2.7 Action Routines

Action routines may (optionally) be associated with any attribute (or tuple field) of any entity or relationship. Up to four separate types of action routine may be associated with each attribute. One action routine will be executed when the attribute value is read (via GetAtt). Another action routine will be executed when the attribute value is written (via PutAtt).

Since an attribute value may be a set, two other action routines may be defined for such set values. One action routine will be executed when a new element is inserted in a set value (via InsertAtt) and the other action routine will be executed when an element is deleted from a set value (via DeleteAtt). In particular, since the value of the tuples attribute is a set of tuples, the Rinset and Rdelete operations will trigger the appropriate action routines, if they exist.

When a "write" action routine is executed, the attribute name, the current attribute value, the record in which the attribute-value pair is contained, and the (requested) new value of the attribute are passed to it as parameters. The value returned by a "write" action routine becomes the new value of the attribute.
When a "read" action routine is executed, the attribute name, the current attribute value, and the record in which the attribute-value pair is contained are passed to it as parameters. The value returned by a "read" action routine becomes the value returned by the GetAtt operation and the value stored in the attribute.

When an "insert" action routine is executed, the attribute name, the current attribute value (set), the new element, and the record in which the attribute-value pair is contained are passed to it as parameters. The value returned by an "insert" action routine becomes the new (set) value of the attribute.

When a "delete" action routine is executed, the attribute name, the current attribute value (set), the element to delete, and the record in which the attribute-value pair is contained are passed to it as parameters. The value returned by a "delete" action routine becomes the new (set) value of the attribute.

Any sequence of entity/relationship operations may be contained in an action routine. In the following example, a "write" action routine associated with the TransientFaults attribute of a processor entity automatically updates the TotalFaults attribute.
UpdateTotalFaults( Ent, Att, CurrentVal, NewVal )
{
    PutAtt( Ent, "TotalFaults",
        GetAtt( Ent, "TotalFaults" ) +
        NewVal - CurrentVal );

    If GetAtt( Ent, "TotalFaults" ) > 100
    Then Message("Entity ", Ent, " is faulty.");

    Return(NewVal);
}

Since PutAtt, GetAtt, InsertAtt and DeleteAtt operations may be called from within action routines, it is possible that circular sequences of calls could occur. To avoid this, special "fast" PutAttF, GetAttF, InsertAttF and DeleteAttF operations are provided. These operations do not activate any action routines associated with the attribute.

4.3 Implementation of the Entity/Relationship Framework

The underlying requirement for the prototype entity/relationship framework is that it represent the information needed for producing and executing adaptable systems. For ease of experimentation, the prototype is implemented in compiled Franz Lisp. It has not been highly tuned for efficiency. Attribute manipulation operations do not check attribute
domains. In many cases, linear search is used where a more sophisticated implementation might have used, for example, hashing. Relationships are implemented in much the same way they appear — as tables of information. It is possible to produce more efficient implementations of relationships by encoding the information into the entities themselves.

Also, no attempt has been made to distill a complete, minimal set of operations on the entities and relationships (as done by Parent [47]). In some cases, there is overlap in the services provided by operations (e.g. Rselect and Rselect1), while in other cases, operations have not been provided because they have not, as yet, been needed (e.g. a relationship-relationship join).

In spite of this, as will be shown in Chapter 10, the performance of the prototype DMS was satisfactory for the tests which were performed.
4.4 Discussion

As mentioned previously, the software adaptation process involves (1) describing allowable changes to the software, (2) monitoring the state of the software, (3) deciding when and how to change the software, and (4) performing the software changes. Large, dynamic systems are too complex for ad-hoc support for these activities. A uniform framework is needed. The RESAS entity/relationship model is a uniform representation framework which provides a basis for the RESAS data management system, which contains the representations of the target system components, a query language which provides access to the data, and an adaptation language which may be used to specify adaptations.

Recent work in artificial intelligence [9, 79], in "second generation" database systems [47], and in programming systems [41] has addressed the need for representation frameworks in which semantic knowledge may be embedded. For adaptable, reliable real-time systems, such semantic knowledge consists of information about component interactions, how the failure of a component affects the states of other components, how the timing characteristics of a component affect the timing characteristics of other components, and how adaptations to a component or its interactions affect other components and the system as a whole.
RESAS needs access to information arising from both the program generation process and the program execution process. The ER model provides a uniform framework which allows the description of the target system components and their interactions, as well as runtime data from monitoring mechanisms within the executing target system.

Given this framework representing the target system, the data manipulation operations described in previous sections provides a language and mechanisms for manipulating the representations of the target system components, and the actual realizations of the components (i.e. the components themselves).

Components and their interactions in an adaptable system may change over time. Action routines may be used to ensure that data and statistics derived from that data, are temporally valid at any given time. For example, a statistic describing the long-term average load placed on a processor by objects assigned to that processor may be computed from monitoring information periodically collected from the executing system. If an object is moved from one processor to another processor, it may be necessary to re-initialize the load statistics of both of the processors, since the underlying assumption - that the load was caused by a specific set of objects - is no longer valid.
Similarly, since a single adaptation may have far-reaching effects, it is necessary that some degree of *consistency* be maintained, both between the representations of different components, and between the representation of a component and the realization of the component. For example, moving an object from one processor to another may involve automatically incrementing the `AvailableMemory` attribute of the former processor and decrementing the `AvailableMemory` attribute of the latter processor.

The ER framework aids the decision-making process by making it possible to observe different *views* of the system. These views may be used to show certain information about the state of the system which is relevant to a particular decision, but hide irrelevant information. For example, a view might show the names and execution times of all objects, and the names and CPU loads of the processors to which they are assigned. It is possible to do complex queries on views. For example: "Find the object on processor P1 that does not communicate with other objects on P1 and, of those, uses the most CPU time". Section 4.6.1 contains examples of useful views.

Since complete knowledge of the state of the executing system is impossible, the representation framework should provide means for handling incomplete information. Default assumptions about component
attributes and component interactions should be representable. The ER framework allows attributes to be given the value "?", representing an unknown value, and adaptation algorithms may use such values in the decision process.

To summarize, the ER representation framework has the following properties:

- It uniformly represents software and hardware components, including object types and instances, software links, processors and hardware communication channels.

- It uniformly represents components at different levels of granularity (e.g. basic and composite objects). Action routines allow the attributes of a component to be represented as functions of the attributes of other components.

- It separates those attributes of a component which are intrinsic to that component (via entities) from those attributes which depend on how the component interacts with other components (via relationships and action routines).

- It represents the current state of components and of their interactions, allowing incomplete information and default assumptions about the components and their interactions to be represented.

- It allows adaptations to be expressed as changes to the states of components and their interactions, and shows how changes to components and interactions affect other components and interactions. Action routines may be used to keep information up-to-date and to maintain consistency both between the representations of different components in the system, and between a component and its representation.

- It allows all interactions between components to be shown explicitly, if necessary.

- It allows the generation of different views which show only the information necessary for making particular decisions.
4.5 Related Research

The relational model [24] provides a formal model of data and addresses important issues, including uniform representation, multiple views and consistency. It provides a well-defined set of data description and query operations. However, the relational model, by itself, does not address the higher level semantics of the data. All relations are simply tables of information. No distinction is made between relations which represent components and those which represent component interactions.

The entity relationship model [12] addresses the need for semantic knowledge embedded in the representation. It may be built on top of a relational model [82]. This type of model seems appropriate for describing the types of information required for an adaptable system. It separates intrinsic attributes of components from those depending on component interactions, it provides multiple views, and a powerful set of operations [47]. The RESAS ER framework is based on the general ER model. Unlike most ER implementations, in the RESAS implementation new copies of tuples are not created by the operations (e.g. Rselect, Rproject), but must be explicitly created via the Copy operation. This causes less overhead and is useful in creating and maintaining views, but care must be taken in ensuring that a relationship is not made invalid by changes to another relationship with which it shares tuples.
Interesting work has been done on temporal database models. These models maintain histories of data values and allow temporal queries on the data. These aspects would be very useful for adaptable systems, particularly for analyzing trends in component interactions and for analyzing the effects of adaptations. However, a complete temporal system (at the current state of the art) would be much too cumbersome to handle the large amount of data generated by an executing target system. The RESAS framework does not support such temporal data. It is, however, possible to use action routines to invalidate old data and gather new data when necessary.

Frame system models have been shown to be useful for representing semantic knowledge about the components of a system. The *action routine* mechanism is a powerful tool which may be used to provide consistency, define default values and, in general, provide a mechanism for describing "procedural" relationships. However, frame systems do not, by themselves, provide a strong set of operations for describing views and complex queries. Much of the semantics of the component interactions is encoded in the action routines, and is not an intrinsic part of the model structure. The RESAS framework thus combines the action routines of frame systems with the semantics of the ER model.
4.6 Mapping Objects to ER Model

Given the entity/relationship representation framework and the object programming model, the components of the target system may be concisely specified. Some entities and relationships used in the RESAS prototype will be discussed in this section.

All object types (basic and composite, concrete and synthetic) and instances are represented as entities. Interfaces are represented as entities, with a relationship representing the connection of interfaces and objects.

For example, the MotionPlanning basic object type from the example in Figure 1 is represented as:

```
ENTITY
Name:                MotionPlanning
Type?Inst:           TYPE
TypeLevel:           CONCRETE
InstLevel:           BASIC
InputRequirements:   (I1)
ExecTime:            29ms
MemNeeded:           8K
MaxTries:            1
```
An interface associated with MotionPlanning is represented as:

**ENTITY**
Name: MPI1
Role: RequestedVelocityVector
Type: VelocityVector6
Direction: IN
SelectionPolicy: FIFO

The OBJECT-INTERFACE relationship records the "ownership" of interfaces by objects, including the fact that MPI1 is an interface of MotionPlanning:

**RELATIONSHIP**
Name: OBJECT-INTERFACE
Tuples:
Object __________ Interface
MotionPlanning __________ MPI1
...

Composite object types are represented in the same way as basic object types, but with three additional relationships.

The CHILD-OBJECTS relationship contains the local names and requested types of the children of a composite type. For example, the System type shown in Figure 16 leads to the following tuples:
RELATIONSHIP
Name: CHILD-OBJECTS
Tuples:

<table>
<thead>
<tr>
<th>Parent</th>
<th>ChildName</th>
<th>RequestedChildType</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>F</td>
<td>Worker</td>
</tr>
<tr>
<td>System</td>
<td>G</td>
<td>Worker</td>
</tr>
<tr>
<td>System</td>
<td>A</td>
<td>Check</td>
</tr>
</tbody>
</table>

The CHILD-CHILD-LINKS relationship records the invocation connections of the children of a composite type. For the System type, this leads to the following tuples:

RELATIONSHIP
Name: CHILD-CHILD-LINKS
Tuples:

<table>
<thead>
<tr>
<th>Parent</th>
<th>Link</th>
<th>Invoker</th>
<th>OutInt</th>
<th>Invokee</th>
<th>InInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>L3</td>
<td>F</td>
<td>Q1</td>
<td>A</td>
<td>I1</td>
</tr>
<tr>
<td>System</td>
<td>L5</td>
<td>A</td>
<td>Q1</td>
<td>F</td>
<td>I2</td>
</tr>
<tr>
<td>System</td>
<td>L4</td>
<td>G</td>
<td>Q1</td>
<td>A</td>
<td>I1</td>
</tr>
<tr>
<td>System</td>
<td>L6</td>
<td>A</td>
<td>Q1</td>
<td>G</td>
<td>I2</td>
</tr>
</tbody>
</table>

The PARENT-CHILD-LINKS relationship represents the connections of parent and child interfaces. For the System type, this leads to the following tuples:
RELATIONSHIP
Name: PARENT-CHILD-LINKS
Tuples:

<table>
<thead>
<tr>
<th>Parent</th>
<th>Link</th>
<th>ParentInt</th>
<th>Child</th>
<th>ChildInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>L1</td>
<td>I1</td>
<td>F</td>
<td>I1</td>
</tr>
<tr>
<td>System</td>
<td>L2</td>
<td>I2</td>
<td>G</td>
<td>I1</td>
</tr>
<tr>
<td>System</td>
<td>L7</td>
<td>Q1</td>
<td>F</td>
<td>Q2</td>
</tr>
<tr>
<td>System</td>
<td>L8</td>
<td>Q2</td>
<td>G</td>
<td>Q2</td>
</tr>
</tbody>
</table>

... 

Synthetic types are represented as entities, and generally contain only a name attribute, although they may contain other attributes describing default information valid for all of their subtypes. Type hierarchies built with synthetic and concrete types are represented by the TYPE-SUBTYPE relationship. It is of the form:

RELATIONSHIP
Name: TYPE-SUBTYPE
Tuples:

<table>
<thead>
<tr>
<th>Type</th>
<th>Subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>BubbleSort</td>
</tr>
<tr>
<td>Sort</td>
<td>QuickSort</td>
</tr>
</tbody>
</table>

The TYPE-INSTANCE relationship represents the connection between all object instances and the (concrete) type from which it was derived. It also records the original (synthetic or concrete) type specified when it was created. This allows the AC to dynamically choose to replace an object instance with a new instance of a different concrete type, but the same specified type. The TYPE-INSTANCE relationship is of the form:
RELATIONSHIP
Name: TYPE-INSTANCE
Tuples:

<table>
<thead>
<tr>
<th>Instance</th>
<th>ChosenType</th>
<th>SpecifiedType</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/Sort1</td>
<td>QuickSort</td>
<td>Sort</td>
</tr>
<tr>
<td>S/Sort2</td>
<td>BubbleSort</td>
<td>Sort</td>
</tr>
</tbody>
</table>

The INVOKES relationship represents the actual connections between basic object instances in the executable program. Its format is:

RELATIONSHIP
Name: INVOKES
Tuples:

<table>
<thead>
<tr>
<th>Invoker</th>
<th>OutInt</th>
<th>Invokee</th>
<th>InInt</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/DA</td>
<td>Q1</td>
<td>S/MP</td>
<td>I1</td>
<td>L1</td>
</tr>
</tbody>
</table>

The PROCESSOR-ASSIGNMENT relationship describes the mapping of basic object instances to the processor nodes. Its format is:

RELATIONSHIP
Name: PROCESSOR-ASSIGNMENT
Tuples:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>S/DA</td>
</tr>
<tr>
<td>P2</td>
<td>S/MP</td>
</tr>
</tbody>
</table>

Hardware components are also represented as entities and relationships. In the prototype target hardware, there are two general types of entity: processors and channels, and one relationship: CONNECTS. For example:
ENTITY
Name: Processor
Type?Inst: TYPE
TotalMem: 256K
CPU: 8086
FloatingPoint?: YES
Faults: 0

ENTITY
Name: Channel
Type?Inst: TYPE
Bandwidth: 10Mbps
Faults: 0

RELATIONSHIP
Name: CONNECTS
Tuples:
Channel__Processor1__Processor2
  C1   P1   P2
  C2   P2   P3
...

For bi-directional channels, the CONNECTS relationship is symmetric with respect to Processor1 and Processor2. While the prototype system does not handle this automatically, it would be possible to specify this for the relationship, so queries would automatically check both directions.
4.6.1 Useful Views

Given the entities and relationships described in the previous section, ER operations may be used to extract information valuable for making adaptation decisions. This section gives some examples.

Make a set of all actual processors:

\[
\text{PROCESSORS} := \text{Rselect(TYPE-INSTANCE,} \\
\text{(ChosenType = "Processor")});
\]

Strip off irrelevant attributes and rename the Instance attribute to be Name:

\[
\text{PROCESSORS} := \text{Rproject(PROCESSORS, Instance, Name)};
\]

Attach the processor's entity attributes:

\[
\text{PROCESSORS} := \text{RjoinER(PROCESSORS, Name)};
\]

Strip off irrelevant attributes:

\[
\text{PROCESSORS} := \text{Rproject(PROCESSORS, Name, TotalMem,} \\
\text{CPU, Faults, nil, nil, nil, nil)};
\]
The result is this relationship:

```
RELATIONSHIP
Name: PROCESSORS
Tuples:
   Name  TotalMem  CPU  Faults
   P1    256K     8086  10
   P2    128K     8086  2
   P3    512K     8085  4
```

With the PROCESSORS relationship, it is possible to answer queries about individual processors or groups of processors. For example:

What is the name of the processor with the most total memory?

```
Biggest := GetAtt(Rmax(PROCESSORS, TotalMem), Name);
```

What is the average number of faults/processor?

```
Avg := Rsum(PROCESSORS, Faults)/Rcount(PROCESSORS);
```

Many other examples are possible. Such views are used in the implementation of the algorithms described in the chapters that follow.
Chapter 5

The Adaptation Control System

The Adaptation Control System (ACS) is composed of the RESAS mechanisms which choose and perform adaptations. It consists of three main components: the monitoring mechanism (MON) which collects data from the executing target system; the adaptation controller (AC) which analyzes the data and decides when it is necessary to adapt the target system, and what adaptations to make; and the adaptation enactment mechanism (AE) which performs the adaptations on the target system. In addition, the data management system (DMS) performs a support role by storing data and providing a uniform data format for the three ACS components. Figure 13 shows the ACS.

Both static and dynamic adaptations involve monitor - choose adaptation - perform adaptation cycles. In the static adaptation cycle, the monitoring mechanism collects data from a current configuration of the target system, and an “off-line” adaptation decision is made,
Figure 13: The RESAS Adaptation Control System
resulting in the off-line construction of a new software configuration. The target system is stopped, the new software configuration is loaded onto the target hardware, and the target system is restarted.

In the dynamic adaptation cycle, the monitoring mechanism, as before, collects data from the current target system configuration. The adaptation controller, running in parallel with the target system, makes adaptation decisions, and the adaptation enactment mechanism performs the adaptations on the target system while it is executing.

In the following sections, each of the ACS components will be discussed.

5.1 The Monitoring Mechanism

The monitoring mechanism is responsible for collecting data from the target system and passing it to the DMS, where the data may be accessed by the other RESAS components. There are two monitoring operations:

- **Sensors** may be embedded within the code of a basic object. The operation *Sense( Attribute)* passes the current value of the object attribute from the target system to the DMS, where it is written into the entity representing the object.
Probes may be initiated by the DMS. The operation GETATTRMP(Object, Activation\textsuperscript{8}, Attribute) causes a request for the attribute's current value to be sent to the target system. Monitoring code embedded within the target system returns the value to the DMS.

The unreliable and parallel nature of the target system hardware may affect the quality of the monitored data. The consistency of monitored data is currently under study by the ISSOS group \cite{ISSOS} and is not discussed here. Questions addressed include: What does consistency mean in a distributed real-time system?; How can consistency constraints be specified?; How can consistency be traded with monitoring performance?

The monitoring mechanism and the data management system are affected by cost and capacity constraints. That is, it is generally not cost-effective to collect and store “complete” information about the state of the target system, and it is generally not possible to retain all past states of the system. Furthermore, a snapshot of the instantaneous state of the target system may be difficult or impossible to obtain \cite{ snapshot}.

In this dissertation, most monitored data is in the form of long

\textsuperscript{8}Activations are discussed in Chapter 7. Briefly, a basic object may be invoked at different times by several other objects. Each resulting activation may have somewhat different attributes (e.g. deadlines). In the following discussion, the activation parameter may be safely equated with the corresponding object.
term averages and counts, so slight inconsistencies do not significantly affect the algorithms and their results. The performance and cost of the prototype monitoring mechanism is discussed in Chapter 10.

5.2 The Adaptation Controller

The adaptation controller is responsible for deciding when to adapt and what adaptation to choose.
The question of when to adapt depends on the goals of the adaptations (e.g. the types of hardware faults to be handled) and on the types of possible adaptations. In some cases, the adaptation controller may notice a distinct event (e.g. a hard processor fault) and take appropriate action at that time. In other cases, adaptations may be performed at fixed times (e.g. periodically). In still other cases, the information which triggers an adaptation may be of a statistical nature (e.g. long-term resource utilization or fault patterns). In those cases, there is a tradeoff between accuracy of the information sample and speed of response. Choosing adaptations based on small samples may produce inappropriate adaptations, while waiting for large samples may delay adaptation unnecessarily, and may invalidate the statistics themselves in some cases.

The ACS and target system is a control system with feedback. Important characteristics of the ACS include:

- **observability.** To what degree can the state of the target system be known by the ACS?

- **controllability.** Can adaptations change the target system from state A to state B?

- **stability.** Can the target system be held in a particular state or set of states?

The input to an adaptation algorithm is information of a certain **scope** (i.e. the portion of the target system from which the information
is drawn), *completeness* (i.e. a measure of the amount of known vs. unknown information), *correctness* (i.e. the degree to which the information is accurate) and *history* (i.e. the length of time for which past values of the information is known). These attributes are essentially orthogonal. For example, it is possible to have information which is complete but incorrect, or is correct but incomplete.

The output of an adaptation algorithm is an adaptation (i.e. state change) which has an *effect* of certain *scope* (i.e. the portion of the target system which will be affected) and *lifetime* (i.e. the length of time the adaptation will affect the target system).

Adaptation algorithms may be categorized by the characteristics of their inputs and outputs (described above), by various measures of their *robustness* (e.g. the degree to which they can handle incomplete information or their sensitivity to small changes in the input information), and by their *planning horizon* (i.e. the length of time into the future for which they can intelligently plan). The planning horizon of the algorithm may not be the same as the lifetime of the effect of the adaptation. For example, an adaptation algorithm may have a short planning horizon and yet the adaptation it produces may have a permanent effect.
When two or more conditions warranting different adaptations arise at the same time, their choice and enactment must be coordinated. The appropriate action depends on the characteristics of the algorithms and their effects. If the scopes of the effects do not overlap, then both adaptations may be performed. Otherwise, one adaptation must be chosen.

Adaptations may be prioritized by the adaptation controller. Priorities may be assigned, for example, by the scopes or lifetimes of the effects, or by the cost of the adaptation. This cost includes the cost of performing the adaptation and the cost to the target system over the lifetime of the adaptation.

Chapters 7, 8 and 9 describe several sample adaptation algorithms. In the tests described in Chapter 10, adaptations are performed periodically.

5.2.1 Fault Modes

In the reliability adaptation algorithms described in this dissertation, processor faults are assumed to be independent and transient and to affect only the activation currently executing. Faults are assumed to not cause latent errors in the programs.
Because the MTTF of the hardware (many hours) is assumed to be much larger than the execution time of the program (several seconds from START to END), multiple independent, transient failures are unlikely and, therefore, only a single fault per application cycle will be considered.

Activations are assumed to be fail-stop. An activation affected by a processor fault will simply abort, producing no output and affecting (by side effect) no other activation. This is a reasonable assumption if the processors are designed with protected memories and if watchdog timers may be used to handle infinite-loop failures.

Fault rates for processors are given in faults per unit of time. The default assumption is that fault rates are zero unless otherwise specified.

Given the hardware and software models presented in this dissertation, there are other possible fault modes. For example, a processor may fail completely, permanently disabling all basic objects (and their activations) assigned to that processor. Interprocessor channels may fail completely, partitioning the hardware (and software). Processor faults may induce errors in the state of an object, disabling all activations associated with the object. The previous assumptions, and the algorithms based on those assumptions, are not invalidated if other fault modes
coexist with the transient fault mode. However, the given sample algorithms may not aid in guarding against these other fault modes, and may, in fact produce adverse effects.

5.3 The Adaptation Enactment Mechanism

Strictly speaking, adaptations are enacted during both the static and dynamic adaptation cycles. In the static case, the enactment consists of compiling, linking and loading a new configuration. This is a relatively straightforward procedure and is not discussed here.

In the dynamic case, the unreliable, real-time and parallel nature of the target system hardware affects the enactment process in several ways:

- Because dynamic adaptation involves making changes to the executing program, it is necessary to maintain consistency of the program while the adaptation is being performed. It is not desirable to allow the program to execute with part of the adaptation performed, but not all. It may be necessary for all objects to agree upon the instant at which an adaptation takes effect.

- The adaptation enactment mechanism must be distributed throughout the target hardware. The imperfect reliability of the target system therefore implies that adaptation enactments may fail. In the event of an adaptation enactment failure, the target system must be returned to a consistent state.
- The real-time nature of the application demands that dynamic adaptations not place a large overhead on the system, nor force the program components to wait for long periods of time or for messages or events which may not occur. The adaptation enactments, both successful and unsuccessful, must be processed without major interruption of the normal flow of the application.
- During an object replacement adaptation, if the replaced object contains state, this must be passed to the new object.

Adaptations are performed by modifying the entities and relationships in the DMS using the PutAtt operation described in Chapter 4. Action routines attached to the entities and relationships send these changes to the appropriate objects in the target system. The XPutAtt(Object, Activation, Attribute, Value) operation may be used within these action routines to send the changes to the target system.

5.3.1 Adaptation Transactions

To address the issues raised in the previous section, adaptations are enacted via adaptation transactions. Adaptation transactions guarantee that the program will not be seen in an inconsistent state by the normal flow of the application during fault-free periods, and that the probability of an inconsistent state visible to the application may be made arbitrarily small in the event of faulty behavior. Adaptation transactions are designed to cause no wait-time overhead. The only overhead is that involved with distributing adaptation commands to the basic objects and performing the actual changes to each basic object.
An adaptation transaction is a collection of adaptations (XPutAtt operations - see Chapter 6) on one or more activations. All of the adaptations within a transaction are given the same, unique transaction identifier, which is an integer counter. The identifier of the first transaction performed is 1, and the counter is incremented after each transaction. In addition, each transaction (and the adaptations associated with it) is given a transaction timeout, which is explained below. The two-phase protocol works as follows:

- **Phase 1:** Each adaptation, along with the transaction's identifier and timeout, is sent from the DMS to the activation. The activation stores the information, but does not perform the change. The activation acknowledges the receipt of each adaptation. If no acknowledgment is received within a time, the adaptation may be re-sent, repeatedly, until an acknowledgment is received or the DMS gives up.

- **Phase 2:** The adaptations are performed. The DMS sends the transaction identifier to the START activation, which includes it in all invocations to the application program (The START activation always sends some transaction identifier, and once it has sent a value X, it may never send a value less than X. It continues to send X until ordered by the DMS to send X+i.). Each activation includes the transaction identifier in all of its invocations.

When each activation is first invoked within an application cycle, it first checks if it has recorded any adaptations with transaction identifiers less than or equal to the identifier it received in its invocation. If so, it processes them in order, from lowest identifier to highest. If the BaseStartTime of the invocation received by the activation is less than or equal to the transaction timeout of the adaptation, then the adaptation is performed. Otherwise, the adaptation is flushed. After all adaptations have been processed, the activation proceeds with its normal function. The second phase can be thought of as
a "wave" of adaptations which precedes a particular application cycle.

If, in the first phase, the DMS does not receive one or more acknowledgments and gives up, it may choose one of two cleanup actions. It may send CANCEL commands to all activations, instructing them to cancel the transaction, or it may wait for the transaction timeout before initiating any other adaptations. If the former action is chosen, then CANCEL commands must be sent to all activations, and acknowledgments must be received from all activations, including the ones which may have been responsible for canceling the transaction in the first place.

If the latter action is taken, then the transaction will be flushed automatically from each activation the first time the activation is scheduled with a BaseStartTime greater than the transaction timeout. However, because an activation always executes any adaptations which have transaction identifiers less than or equal to the one transferred in the program invocations, no newer (i.e. higher numbered) transactions may be performed until the canceled transaction's activation deadline has expired.

In the second phase, since all adaptations have been successfully loaded into the activations, the transaction identifier "bubbling" down
through the invocation graph will activate the adaptations in all activations with which it comes into contact. If, due to failure, one part of the graph is temporarily cut off from the invocation, those activations will be processed on the next, non-faulty execution of the graph. The only danger of inconsistency arises if the part of the graph continues to be "dead" past the transaction timeout (that is, if NO good invocation reaches that part of the graph before the timeout). In this case, the activations in the active part of the graph will have performed the adaptations, while those in the dead part of the graph, if it is ever reactivated, will flush those adaptations. For this reason, it is necessary to choose transaction timeouts which are long enough so that the probability of such extended down-time is remote.

Since the START activation does not actually "exist" in the target program, it is necessary for the AE mechanism to send the new transaction id to each of the activations invoked by START. If there is more than one such activation, an additional level of transaction mechanism is needed to ensure that all such activations agree on the current transaction id. In effect, therefore, the adaptation transaction mechanism described above ensures that for a given composite object, if the transaction id can be atomically passed to all "child" basic objects connected to the composite object's input interfaces, then all basic objects contained within the composite object will correctly process the adaptation transaction.
As noted earlier, during an object replacement adaptation, if the replaced object contains state, this must be passed to the new object. The RESAS prototype copies the entire state area from one object to the other with no understanding of the semantics of the state variables. This copying is done during Phase 2 of the adaptation transaction which performs the object replacement. Alternative techniques and related issues have been explored by other researchers (e.g. forwarding [13] and standard forms [74]).

5.3.2 Adaptation Transactions and the DMS

Adaptation transactions allow changes to the program representation in the DMS to be correctly enacted on the target system. Adaptation algorithms manipulate the DMS representation and the changes are passed to the adaptation enactment mechanisms embedded in each object in the target system.

Consider the modification of the MaxTries attribute of an activation. The DMS operation is:

```
PutAtt( Activation, MaxTries, 2);
```

This operation activates the following action routine, associated with the MaxTries attribute of the activation:
Begin;

First, calculate a new StartTime and Deadline. Then...

Id = BeginAdaptationTransaction(Timeout);

If XPutAtt(Activation, MaxTries, 2) and
   XPutAtt(Activation, StartTime, 12345) and
   XPutAtt(Activation, Deadline, 54321)

Then If EndAdaptationTransaction(Id)
   Then goto Commit;
   Else goto Abort;

Else Wait(Timeout);
   goto Abort;

Commit: /* Successful. Update the DMS */
   PutAttF(Activation, MaxTries, 2);
   PutAttF(Activation, StartTime, 12345);
   PutAttF(Activation, Deadline, 54321);
   return(Success);

Abort:
   return(Failure);

End;

There are three possible outcomes of an adaptation transaction:

- An XPutAtt operation may fail to get a reply from the target object within a specified time. In this case, the transaction aborts and waits for the transaction to flush itself.

- All XPutAtt operations may complete successfully, but the EndAdaptationTransaction command, which initiates Phase 2, may arrive at the target system after the transaction timeout, in which case the transaction has already flushed itself. In this case, the transaction aborts.
The XPutAtt operations may complete successfully and the EndAdaptationTransaction command may arrive at the target system before the transaction timeout, in which case, the transaction commits and updates the DMS.

Chapter 10 examines the performance and overheads of the monitoring and adaptation enactment mechanisms.
Chapter 6
Implementation of the RESAS Prototype

6.1 Introduction

This chapter addresses the RESAS mechanisms which execute, monitor and adapt the software.

6.2 The Hardware

The test hardware configuration is shown in Figure 14. It consists of a Sun 3/50 workstation, an Intel SeriesIV development system and an Intel 8086-based multiprocessor. SeriesIV-to-multiprocessor connection is via a parallel channel (8 bits each direction) and the other connections are via 9600 baud serial channels.
The Sun workstation, executing Franz Lisp, is the site of the DMS, the AC and all communication with the application programmer. The application programmer uses the workstation to create, modify and observe the execution of the application software (application program I/O is actually directed to a separate terminal attached to the multiprocessor, although it could be directed to a window on the Sun if the Sun-to-multiprocessor link bandwidth were higher).
When a program has been configured, it is loaded from the Sun onto the SeriesIV. The SeriesIV, executing Intel software development tools, is used to compile and link the software, and to load it onto the multiprocessor.

The multiprocessor consists of 8 Intel 86/30 single-board computers (SBC's) connected by a Multibus. Each SBC includes an 8086 microprocessor (8 Mhz), an 8087 floating point co-processor, and 256K bytes of RAM. Of each SBC's 256K, 192K is addressable only locally (i.e. by the 8086/8087 on the SBC). The remaining 64K is addressable both locally and remotely (i.e. by other SBC's on the Multibus). As a result, each SBC has access to 192K of local memory and a total of 512K (8×64K) of "shared" memory, addressable by all SBC's on the Multibus.

One processor, P0, of the multiprocessor is dedicated to providing an interface to the DMS on the Sun workstation. The (only) object on P0 is seen, by other objects in the system, as a stand-in for the DMS. Thus, all application objects in the system see only other objects and their operations.
6.3 The GEM Operating System

The runtime system of the multiprocessor is based on the GEM real-time operating system, upon which an object-based extension has been built. Details of GEM are given in [58]. For the purposes of this discussion, it is only necessary to understand that applications are constructed as a set of objects which encapsulate all data and compute functions using the data. Each object is served by some number of GEM processes, chosen at object creation time. Processes are not shared between objects.

All active processes on an SBC compete for the processing cycles based on their priorities and deadlines. The GEM process scheduling queue on each SBC is “priority-major”. That is, there are 8 priority levels (from 1 (low) to 8 (high)) and within each priority level, processes are arranged in deadline order (and if the priorities and deadlines of two processes are identical, they are arranged in FIFO order). The highest priority process with the nearest deadline is always run to completion.
6.4 Generating Objects

6.4.1 RESAS Objects

Figure 15 shows a typical RESAS basic object "A". It contains three input interfaces and two output interfaces. Assume A's input requirements specification is "(I1 and I2) or (I1 and I3)". At least two of the three input interfaces must therefore be invoked before A is activated, computes its function and invokes the two output interfaces. In addition to the "application" interfaces (I1, I2, I3, Q1 and Q2) shown in the figure, A contains a pair of interfaces (I0, Q0) which are used for adaptation operations.

Input buffers associated with each input interface store parameters received during invocations and flags describing the status of the input interface. For each input interface, there is (potentially) one buffer for each possible activation of the object.

Output buffers associated with each output interface store parameters to be sent during invocations and flags describing the status of the output interface. For each output interface, there is (potentially) one buffer for each possible activation of the object.
Figure 15: A Typical RESAS Basic Object
Attributes and static local variables comprise the object state. There is one copy of each local variable per object. Local variables are visible only within the object. There is one copy of each attribute per activation. This is because adaptations are written in terms of activations, and therefore the data is represented accordingly. It is possible to represent object attributes by using action routines in the DMS to compose object attribute values from the values of attributes of the object's activations.

In addition to the object state, each GEM process allows dynamic variables to be created on its process stack. The lifetime of a dynamic variable is one activation execution.

An object contains three code components. The application code is written by the application programmer. It is the actual function computed by the object. It is of the form:

\[
\text{Begin;}
\]

\[
\text{Accept(I1,Parameters,Status);}
\]
\[
\text{Accept(I2,Parameters,Status);}
\]
\[
\text{Accept(I3,Parameters,Status);}
\]

\[
\text{compute function of inputs and object state...}
\]
\[
\text{Send(Q1,Parameters);}
\]
\[
\text{Send(Q2,Parameters);}
\]
\[
\text{End;}
\]
The adaptation code is responsible for the handling of object attributes. It performs several functions:

- It provides the Sense(Attribute) operation to the object's application code. This operation causes the value of the attribute to be sent to the DMS (via the object's Q0 output interface).

- It provides the GetAtt(Attribute) operation to the DMS. The GetAtt operation is activated via an invocation from the DMS to the object's I0 input interface and causes the attribute's value to be sent to the DMS (as with the Sense operation).

- It provides the PutAtt(Attribute, Value) operation to the DMS. The PutAtt operation is activated via an invocation from the DMS to the object's I0 input interface and causes the attribute's value to be updated to the new value sent from the DMS. A reply invocation is sent from the object (via Q0) to the DMS when the update is complete.

- It provides the XPutAtt(Attribute, Value, TransactionId, Timeout) operation to the DMS. This operation is similar to the PutAtt operation, but is used to implement the adaptation transaction mechanism described in Chapter 9.

Although Sense, GetAtt, PutAtt and XPutAtt are implemented as special-purpose operations, they may be considered to behave in the same way as other invocations.

The scheduling code is responsible for determining what action to take when an invocation is received at an input interface. The pseudocode for a scheduler is shown below. The BaseStartTime is the time at which the START object initiated a particular application cycle. The StartTime is the latest time (offset from each BaseStartTime) at which the activation can start and still meet its deadline.
Begin;

/* All variables are on a per-activation basis. */

Loop Forever;

Sleep until awakened by an invocation on some input interface;

/* Record the invocation */

Case:

This is the first interface invoked with this BaseStartTime:
{
    Reset all flags;
    Mark this input interface as "invoked";
    Copy the input parameters into an input buffer;
    Perform any adaptations which are activated;
    Break;
}

The activation has already fired for this BaseStartTime:
{
    Return;
}

Default: /* Valid invocation, not the first. */
{
    Mark this input interface as "invoked";
    Copy the input parameters into an input buffer;
    Break;
}

EndCase;
/* Check if sufficient inputs have been invoked to cause the activation to be executed. */

Case:

All input interfaces have been marked "invoked":
{  
  Break;
}

The Input Requirements Specification has been satisfied:
{  
  Sleep until StartTime + BaseStartTime;
  Break;
}

Default: /* Insufficient inputs ready. */
{  
  Return;
}

EndCase;

/* At this point, sufficient inputs are ready. */

If this activation has not yet fired for this BaseStartTime and no new set of invocations has arrived:

Then Execute the application code;

EndLoop;

End;

The action of object A proceeds as follows. At startup time, one process belonging to A is awakened, initializes local data, including object attributes and runtime system flags, and goes to sleep. All processes then begin to execute the scheduling code.
Suppose that an output interface of object B is connected to an input interface of object A. B may invoke its output interface, using an invocation of the form:

\[ \text{Send(OutputInterface,Parameters)}; \]

Using information from the ACTIVATION-INVOKES relationship (stored within the invoker’s attributes), the Send operation is translated into an invocation command of the form:

\[ \text{Invoke(Object,InputInterface,Activation, BaseStartTime,TransactionId, Parameters)}; \]

*Object* is the name of the invoked object (in this case “A”). *InputInterface* is the name of the input interface of A which is attached to the *OutputInterface* of B. *Activation* is the activation of A which is “connected” to the activation of B which issued the Send operation. *BaseStartTime* is passed along from one activation to the next. *TransactionId* is used by the adaptation code and will be discussed in Chapter 9.

The invocation mechanism chooses an idle process from A’s list of server processes and wakes the process. It passes the process information about which input interface has been invoked and where to find the parameters. The process awakens and begins execution of the scheduling code shown above.
This mechanism operates on activations (not objects) and their invocations, since the flags and variables (e.g. StartTime) consist of one copy per activation in the object. Therefore, it is possible for more than one activation to be executing at a time. The BaseStartTime carried along with each invocation is a unique identifier of the set of invocations to which it belongs. If an invocation from one set should “overtake” an invocation from a previous set, the former will clear out the latter. Therefore, programs must be written so that this cannot happen (using explicit synchronization).

If all input interfaces of an activation have been invoked, the activation may be executed immediately. The priority of the application code is lower than that of the scheduling code, so invocations are recorded “immediately”, possibly at the expense of the execution of application code. All application code executions on a processor are scheduled on a shortest-deadline-first basis and are preemptable.

If some sufficient subset of activation inputs are ready (in this example, two of the three - this is from the input requirements specification described in Chapter 3), the activation waits until the last possible time (StartTime) it may start and still complete by its deadline. It is hoped that the remaining inputs will become ready before that time. If not, the activation will be executed with those inputs which are ready.
When an application code execution completes, it then sends invocations to all output interfaces (and the activations attached to them), and the cycle continues.

The number of processes assigned to an object depends on the number of activations which may be running in parallel and upon the number of input interfaces (processes) which could be waiting for StartTime. An upper bound on the number of processes needed by an object is:

\[
\text{NumberOfActivations} \times \text{NumberOfInputInterfaces}
\]

Chapter 10 describes timing measurements done on the RESAS mechanisms and examines the effect of the mechanism overheads on the application program.
Chapter 7
Generation of the Initial Configuration

7.1 Introduction

As described in Chapter 3, the application programmer designs and describes object types, both composite and basic, concrete and synthetic. The actual program is composed of instances of these types, and in particular, the executable program is made up of instances of basic types.

This chapter describes the object instantiation process, the analysis of the real-time and reliability characteristics and requirements, and sample algorithms used to generate an executable program.
7.2 Instantiating Objects

The programmer's description of a target program looks like that in Figure 16. START and END objects have been connected to the composite object type System.

When a request is made to instantiate a type, if it is a concrete type, an object instance is created. If it is a synthetic type, the adaptation controller (AC) may choose to instantiate any concrete type in the type hierarchy with the requested type as root node. In the initial configuration generation (ICG) phase, the concrete type chosen is always one with the least execution time. This allows the initial processor assignment to be done most easily.

When a composite object type is instantiated, the child types are instantiated and so on, recursively, until basic object types are instantiated. Concrete composite types may have synthetic types as child objects. If a request is made, at any level, to instantiate a synthetic type, the adaptation controller chooses an appropriate concrete type.

Consider the example in Figure 16. The System type is a composite type with three child objects. Two of the child objects are of type Worker and one is of type Check. The internal structures of the
child objects are not visible. It is not possible to see, from the System
type description, whether the Worker type is composite or basic. The
Worker type is a composite type with two child objects - one of type
Pass1 and one of type Pass2. Types Check, Pass1 and Pass2 are basic
types.

Figure 17 shows the result of instantiating an object named S of
type System. The three child objects are instantiated. Because type
Worker is composite, its child objects are instantiated as well. All inter­
face connections are made as specified in the type descriptions.

Figure 18 shows the result of resolving S down to its basic objects.
These basic object instances correspond to the actual software which will
be executed. The intermediate object layers (i.e. F and G) are stripped
away and the multi-link invocation paths between basic object interfaces
are combined into single links.

The information in all three figures is retained by the DMS during
execution of the actual software. This allows the software to be adapted
at all levels.
Figure 16: Composite and Basic Types
Figure 17: Instantiating Object S of Type System
Figure 18: Resolving Object $S$ of Type System
7.3 Activations

The application programmer views the target software as a collection of object types which are instantiated by RESAS. The RESAS mechanisms which produce and execute the target software view it as a collection of basic object instances. In order to analyze the real-time performance and reliability of the target software, RESAS algorithms need an additional view of the target software: as a collection of activations.

Figure 18 shows a basic object invocation graph resulting from the instantiation of a program. Object S/A is invoked by both object S/F/D and object S/G/D, but as can be seen from the figure, S/A is actually activated twice. Therefore, the characteristics of interest, including deadlines and reliability information, may have separate values for each of the two activations. For example, the first activation (called S/A_0) may have a short deadline so that object B can begin quickly, while the deadline of the second activation (called S/A_1) may be somewhat longer.

The general problem may be stated in terms of "shared servers": What should determine the required characteristics of an object which is asynchronously invoked by a number of other objects? In general, the performance and reliability characteristics required of an object are
determined by the object's position in the program's "invocation graph" (i.e. when, and by what objects, it is invoked and when, and what objects it must invoke). Therefore, if an object is invoked by more than one other object, it will be activated more than once and will "appear" more than once in an acyclic invocation graph. In many situations, however, objects are not "shared" and each basic object instance corresponds to exactly one activation.

To properly analyze the real-time performance and reliability of the target program, it is necessary to "unwind" the invocation graph in Figure 18 into the acyclic graph shown in Figure 19. This figure shows each activation of each object explicitly. As will be seen, representations of these activations must be maintained during the entire lifetime of the system.

7.4 Instantiating Activations

The InstantiateActivation algorithm performs the "unwinding" of the basic objects into activations. It is essentially a simulation of the execution of the object invocation graph. First, the START activation is created from the START object by copying the attributes of the START object. Then the call InstantiateActivation(START_0) is made. The algorithm is shown below:
Figure 19: Activations and Their Invocations
InstantiateActivation(Act_X)
Begin

Let "Object" be the object from which Act_X
was created;

For each output interface "QI" of Object
Begin
Let "Invokee" be the object which is connected
to QI;

Mark as "invoked" the input interface "II"
of Invokee which is connected to QI;

If ALL input interfaces of Invokee have
been marked "invoked"
Then
Begin
Increment Y, the activation counter for Invokee;
Create activation "Invokee_Y"
from object Invokee;
Insert the tuple
((object Invokee)(activation Invokee_Y)...) 
into the OBJECT-ACTIVATION relationship;
Insert the tuple
((invoker Act_X)(output QI) 
  (invokee Invokee_Y)(input II)...) 
into the ACTIVATION-INVOKES relationship;
Reset to "not-invoked" all input
interfaces of Invokee;
InstantiateActivation(Invokee_Y);
End;
End;
End;

The result of this algorithm is an acyclic invocation graph, with
nodes consisting of activations, and invocation arcs described by the
ACTIVATION-INVOKES relationship.
7.5 Timing Analysis

In addition to creating the activation graph, the InstantiateActivation algorithm may also provide estimates of start and end times for the activations. The execution times of the activations are assumed to be known. If upper and lower bounds for invocation times are known, then upper and lower bounds on the start- and end-times of the activations may be calculated as follows.

As each activation \( X \) is instantiated,

\[
X.EarliestStartTime = \max_{I} \left[ I.EarliestEndTime + \text{MinInvocationTime} \right] 
\]

over all invokers \( I \) of \( X \)

and

\[
X.EarliestEndTime = X.EarliestStartTime + X.ExecutionTime
\]

Similarly,

\[
X.LatestStartTime = \max_{I} \left[ I.LatestEndTime + \text{MaxInvocationTime} \right] 
\]

over all invokers \( I \) of \( X \)

and

\[
X.LatestEndTime = X.LatestStarttime + X.ExecutionTime
\]
Invocation times are taken with respect to the size of the invocation message. In the prototype target hardware, they are assumed to be approximated by a linear function, with a constant "overhead" term plus a term related to the size (in bytes) of the invocation message. For example, for the prototype hardware configuration, an invocation of $N$ bytes between processors takes $(0.0027 \times N + 5.2)$ milliseconds.

If there is a significant difference between the maximum and minimum invocation times for a particular hardware configuration, the estimated earliest and latest start- and end-times may be loose bounds, and therefore not very useful. If, on the other hand, the invocation times are relatively uniform (as they are for the prototype target hardware), the estimates may be used as approximations to the actual start- and end-times of the activations.

Other types of timing analysis are possible with RESAS objects. For example, the invocation rate of each object is a product of the number of activations associated with the object and the rate at which the START object issues invocations. The "steady state" amount of processing power needed by an object is the product of its invocation rate and its execution time. This may be used by processor assignment algorithms, as discussed in [10].
7.6 Service Calculation

In order to make decisions about the reliability of the program as a whole, some measure of the amount of service provided by each activation is necessary. That is, some activations may be more important for the overall system function than others. The service value provided by an activation is a number $0 \leq S \leq 1$. It may have different meanings, depending on the application:

- It may be a measure of the likelihood that the system requires the activation for a particular situation. For example, a system with two activations providing the same service may assign a service value of 0.5 to each of them, if they are each used half of the time.

- It may be a measure of the quality of the service provided by the activation. If, for example, two activations provide similar services, but one provides a more accurate result than the other, the service value of the former may be somewhat higher than the value of the latter.

The underlying point is that, as RESAS attempts to generate more reliable configurations, it will try to preserve the activations with higher service values, possibly at the expense of those with lower values.

Since the experiments in this dissertation assume single-activation failures, the service values chosen here will be calculated as follows:

- Each of the END activations will be assigned a fraction of the total service (summing to 1). In the example in Figure 18, an END attached to one of the manipulator paths might be given a service value of 0.6, while the other END might have a service value of 0.4.
The service values of the other activations will be chosen by calculating the amount of service lost (at the END's) if the activation fails.

For example, if the failure of an activation X would cause both manipulators to become inactive, then the service value of X is 1. If the failure of X causes one of the manipulators to fail, then the service value of X is 0.6 (or 0.4), and so on. Figure 20 shows the components of the example in Figure 18, with possible service values.

The service calculation algorithm is as follows:
Figure 20: Service Values
CalculateService(ActX)
Begin
Mark ActX as "dead";
Killed(ActX);

service = 0;
For each END
   If END is "dead"
      Then service = service + service(END);

return(service);
End;

Killed(ActX)
Begin
For each invokee I of ActX
   Begin
      Mark I's corresponding input interface as "dead";
      If a sufficient number of I's inputs are dead,
         so that I cannot "fire"
         Then Begin
            Mark I as "dead";
            Killed(I);
         End;
   End;
End;

The result of this algorithm is that, if there is only one END, then all activations that are "weak links" in the sense that their lone failure will cause complete loss of service have service value 1 and all other activations have service value 0. This is consistent with the processor fault assumptions, described in Section 5.2.1, that only single, transient faults will be considered, since only a single activation fail will be caused by
each processor fault, and this activation failure will either cause complete loss of service or no loss of service. In the worst case, for N activations this algorithm has time complexity $O(N^2)$, since the failure of an activation may potentially affect up to N-1 other activations which lie between it and an END activation.

7.7 Processor Assignment

Activation invocation graphs are related to a form of graphs called activity networks. Various analysis and assignment algorithms (e.g. PERT and CPM [26, 78]) for activity networks have been in use for many years. The general problem of assigning activity networks to processors is NP-complete. Heuristics have been proposed for such assignment problems [83, 48]. The following algorithm is adapted from the algorithm described in [59].

The PROCESSOR-ASSIGNMENT relationship describes the mapping of basic object instances to processor nodes. The real-time performance and reliability characteristics of the target software are most easily manipulable using the activation view of the software. Unfortunately, activations are not physical entities. More than one activation may correspond to a single basic object, and it is the basic object which
must be assigned to a processor. The algorithm described here assigns basic objects to processors, but does so by considering them to be groups of buddy activations corresponding to basic objects.

If a basic object has only one activation associated with it, then the goals (deadline, reliability) of the activation are the goals of the object. If, however, a basic object has more than one activation associated with it, the goals of the activations may conflict and must be dealt with. For example, if object X has two activations X_0 and X_1, it is possible that X_0 has an "affinity" for processor P0, while X_1 has an affinity for P1. Since X may only be mapped to one of the processors, a choice must be made. This problem will be dealt with in the following sections.

7.7.1 The Processor Assignment Algorithm

The "outer loop" of the processor assignment algorithm is a bin-packing algorithm [59] which, while not optimal (since the problem is NP-complete), has been shown to be good in many circumstances.
Because the processor assignment is not known in advance (obviously), the actual invocation times between the activations are not known, and therefore the actual start- and end-times are not known. These must be computed as the algorithm progresses. The latest- and earliest-start-times are used as estimates. The algorithm shell is as follows:
Place the activations in a full-order which includes the invocation-induced partial order in reverse. That is, place the activations in an order so that an activation precedes its invokers and succeeds its invokees. Thus, the START_0 activation will be last in the order.

For each activation "ActX" in the order (ignoring START and END activations)

Begin

For each processor P

Begin

GetDeadline(P,ActX);
ActualStartTime = FitInSchedule(P,ActX);
If ActualStartTime >= ActX.LatestStartTime
Then Place P in EligibleSet;
Else
    If ActualStartTime >= ActX.EarliestStartTime
        Then Place P in PossiblyEligibleSet;
        Else Place P in InEligibleSet;
End;

ChosenProcessor =
ChooseRelProcessor(ActX,EligibleSet);

If no ChosenProcessor
Then ChosenProcessor =
    ChooseFastProcessor(ActX,PossiblyEligibleSet);

If no ChosenProcessor
Then ERROR;
Else
Rinsert(PROCESSOR-ASSIGNMENT,
    (processor ChosenProcessor)(activation ActX));
End;

The functions GetDeadline, FitInSchedule, ChooseRelProcessor and ChooseFastProcessor are described below.
An "eligible" processor is one which, no matter what the processor assignments of ActX's invokers are, will guarantee sufficient processing time between the latest possible time at which ActX may be invoked and ActX's deadline so that ActX can complete. An "possibly-eligible" processor cannot guarantee this property, but for some assignments of ActX's invokers it will hold. An "ineligible" processor cannot provide sufficient processing time in any circumstances. Thus, the algorithm chooses a reliable eligible processor if one exists, otherwise, it "gambles" with the possibly-eligible processor with least likelihood of causing missed deadlines. This algorithm has worst case time complexity of $O(N \cdot M)$ for $N$ activations and $M$ processors.

7.7.2 Deadline Calculation

The GetDeadline function calculates the actual deadline needed by ActX, based on the StartTimes of ActX's invokees (which have already been assigned to processors because of the ordering) and the invocation times of the links between the processors to which the invokees have been assigned, and the processor P which is being tried for ActX.
GetDeadline(P, ActX)  
Begin  
Deadline =  
MIN [ StartTime(ActI) -  
Invocation time of ActI by ActX ]  
over all invokees ActI of ActX  
End;  

7.7.3 Schedule Generation  

The FitInSchedule operation is the bin-packing algorithm which allows activations to be preempted. Each processor maintains a schedule relationship containing all of the activations already assigned to it and their starttimes and deadlines. For example:  

RELATIONSHIP  
Name: ProcessorISchedule  
Tuples:  
activation  start  deadline  
ActX, 290  330  
ActY, 250  290  
ActX, 200  250  
ActZ, 140  180  

An activation (e.g. ActX) may be preempted (e.g. by ActY) in the schedule. This is due to the fact that, while the assignment order guarantees that an activation is never assigned until after all of its invokees have been assigned, there is no guarantee on the order in which two activations which are neither ancestors nor descendants of each
other will be assigned (and in fact, no a priori time-order can be given, since the times depend on the invocation times between the interprocessor links).

The FitInSchedule algorithm is as follows:

FitInSchedule(P, ActX)
Begin

Starting at Deadline(ActX), work backward through processor P's schedule.

At each empty interval, place a slot which includes, if possible, all of the execution time of ActX.

If the interval is too small to contain all of ActX, place as much of ActX there as possible, and continue through the schedule until the remainder of ActX is placed into the schedule.

Return the start time of the earliest slot for ActX (i.e. the overall start time of ActX)

End;

The actual schedules generated by this algorithm could be executed by a suitable runtime system. However, as described in Chapter 6, the prototype RESAS target hardware does not execute fixed schedules, so these schedules cannot be guaranteed to be followed exactly. The schedules are used only to estimate the feasibility of assigning an activation to a particular processor.
7.7.4 The Processor Choice Algorithms

The bin-packing algorithm may leave several eligible processors to choose from. In these cases, an algorithm is called to choose a processor from those eligible processors.

The reliability algorithm is as follows:

ChooseRelProcessor(ActX, SetOfProcessors)
Begin
Choose the processor P in SetOfProcessors with:
MAX[ StartTime / P.FaultRate ]
where StartTime is the start time of ActX when it is inserted into P's schedule.
{If P.FaultRate is 0, it is set to 1.}
End;

The effect of this algorithm is that, if processors have approximately the same fault rates, then ActX is assigned to the one which would allow the latest start time. Conversely, if processors allow similar start times, then ActX is assigned to the one with the lowest fault rate.

If only "possibly-eligible" processors are available for ActX, then the processor allowing ActX the latest possible start time is chosen, using the ChooseFastProcessor algorithm:
ChooseFastProcessor(ActX, SetOfProcessors) 
Begin

Choose the processor P in SetOfProcessors with:

MAX[ StartTime ]

where StartTime is the start time of ActX when it is inserted into P's schedule.

End;

Since the default fault rate for processors is zero, ChooseRelProcessor is the same as ChooseFastProcessor in the default case.

7.7.5 Assigning Multi-Activation Objects

While the processor assignment algorithm described above is written in terms of activations, the basic objects corresponding to the activations are actually assigned to the processors. If a basic object unwinds to only one activation, the algorithm may be used as it is shown. However, if a basic object unwinds to more than one activation, the assignment decision becomes more difficult. This is particularly true since the activations are assigned in a specific order, with the invokees of an activation assigned before the activation itself. If this order is disrupted, as it may be with multi-activation objects, it is necessary to estimate the deadlines of those activations for which all invokees have not yet been assigned.
As the assignment algorithm is proceeding, suppose an activation is encountered which, along with its other buddies, all correspond to the same object. The original activation may be assigned as usual, but it is possible that some of the other buddies may not fit on the processor. This cannot be known with certainty until all of the buddies have been "reached" by the normal flow of the algorithm.

Therefore, when some buddy is first encountered by the assignment algorithm, all of the buddies are tentatively assigned to a processor together. The deadline of each buddy is calculated as mentioned before, but if an invokee of a buddy has not been assigned, that invokee's earliest start time is used.

Each buddy attempts to fit into the schedule of the processors. As the buddy assignment proceeds, the sets of "eligible" and "possibly-eligible" processors are intersected so that, after all buddies have been processed, the set of "eligible" processors contains all processors which are "eligible" for all buddies and the set of "possibly-eligible" processors contains all processors which are "possibly-eligible" for some buddies and not "ineligible" for any buddy.

The reliability algorithm is used to choose from these processors as before. The metric value considered is now the sum of the metric values
for all buddies. Given the independent, transient failure model, the sum is the appropriate function, since the probability of multiple failures of the same object within a single execution of the target software is low. All buddies are then assigned to the chosen processor.

Later in the activation assignment, each buddy will eventually be encountered again. When this happens, the buddy is removed from the processor's schedule and reinserted so that it may, possibly, be shifted to a more appropriate spot.

As noted previously, this is not an optimal algorithm. It is possible that, when the final activation (actually the activation invoked by the "START_0" activation) is assigned, there may be no "eligible" or "possibly-eligible" processor for it. In this case, the algorithm fails. It would be possible to do some heuristic shifting of assignments, or even an exhaustive search of all assignments (although the cost would be prohibitive), but this has not been implemented. In the RESAS prototype, if the algorithm fails, the application programmer is consulted for suggestions.
7.7.6 Schedule Stretching

The assignment algorithm described in the previous sections produces schedules which are skewed toward the overall deadline of the program. That is, there may be long gaps in the schedules near the time=0 end and no gaps near the time=deadline end. This wouldn't matter if the goal were just to make a valid schedule. However, because the software is meant to be adaptable, it is advantageous to have spread-out schedules with gaps around activations, so that changes to the execution times of activations may be made without major changes to the schedules.

Therefore, after all of the assignments are made, the schedules are stretched back toward 0. If the overall start time of the program is S and the overall deadline is D, then the deadlines of all activations are recalculated as follows:

\[
\text{StretchFactor} = \frac{D}{D-S}
\]

\[
\text{NewDeadline} = D - \text{StretchFactor} \times (D - \text{OldDeadline})
\]

If there is more than one overall deadline (i.e. if there is more than one END object), the minimum StretchFactor is used.
7.8 An Example

Returning to the example of Section 3.5, instantiation of the System type produces an instance S with instances of the child object types: S/DA of type DataAcquisition, S/MP of type MotionPlanning, and so on. During the initial configuration generation process, all objects' MaxTries attributes are set to one. If a synthetic object type is specified (as with AnyForceTransformation), the fastest type (the ForceTransformation basic type) is chosen.

Activations are generated from the basic objects. In this program, each basic object has only one associated activation. Activation S/DA_0 is created from D/DA, S/MP_0 from S/MP, and so on.

Next, the service values of the activations are calculated. Since there is only one END object (with service value 1.0), and every object is necessary for the successful invocation of the END object, all activations have service value 1.0.

Finally, the activations (and their associated basic objects), starting with S/OC_0, are assigned to processors. After stretching, the schedules are:
RELATIONSHIP
Name: Processor3Schedule
Tuples:
activation  start  deadline
S/IJ_0     1360    1450
S/FT_0     1206    1226
S/MP_0     203     348

RELATIONSHIP
Name: Processor4Schedule
Tuples:
activation  start  deadline
S/IL_0     1468    1483
S/OL_0     1257    1292
S/DK_0     1070    1185
S/JA_0     524     779

RELATIONSHIP
Name: Processor5Schedule
Tuples:
activation  start  deadline
S/OC_0     1495    1500
S/ID_0     1050    1450
S/JC_0     213     348
S/DA_0     39      99

The configuration is compiled, linked and loaded onto the target hardware as described in Chapter 6.
Chapter 8
Iterative Static Adaptation

8.1 Introduction

Typically, an initial configuration is generated with little knowledge of the reliability and performance attributes of either the software or the hardware components. This is not due so much to a weakness in the design process as it is to the lack of raw information about the needs and capabilities of the components and of the application domain.

Information used in the Initial Configuration Generation (ICG) process may be based on default assumptions, such as the assumption that all processors have the same failure rate. Once the initial configuration has been generated, it may be tested on the actual hardware with either seeded or natural failure processes (depending on the natural failure rates for components, which may range from MTTF’s of several
hours to many thousands of hours). Failure modes may be seeded, particularly those which are assumed to be relatively common or dangerous. The results of these tests are more data on the performance and reliability attributes of the software and hardware. Using these data, the ICG algorithms (and/or other algorithms) may be rerun, resulting in a new configuration which may be tested, and so on.

This iterative static adaptation cycle may be repeated as often as is necessary during the design and implementation phase to give programmers a good understanding of, and confidence in, the software. This interleaving of design and implementation is sometimes called experimental programming [5]. Once the software has been released, the cycle may be repeated occasionally, to allow the system to adapt to changing environmental conditions. The following sections describe the iterative static cycle and the mechanisms which implement it.

8.2 The Static Cycle

The static adaptation cycle consists of the following steps:
1. Generate a configuration using the available data and the ICG algorithms (described in Chapter 7) and possibly "hints" from step 5.
2. Load the configuration and initialize the appropriate attributes.
3. Execute the configuration and collect data.

4. After some time, stop the monitoring process and compile statistics.

5. Execute the adaptation choice algorithms (described below), producing "hints" of particular changes to make to the current configuration.

6. Go to step 1.

Step 5 is optional. It may be useful to simply rerun the ICG algorithms in step 1 with the newly collected data, so that the updated configuration more closely matches the actual (or seeded) failure assumptions.

The point at which step 4 is executed depends on the nature of the application and on the failure rates and assumptions. In general, it may be done when sufficient data has been collected so that statistics on failure rates and environmental characteristics are accurate, but not too infrequently, so that the statistics do not hide periodic changes or the adaptations are too infrequent.
8.3 Hints

As described in Chapter 7, the initial configuration is created using default information about the software and hardware, and default choices are made, for example, when choosing a particular concrete type to use for a specified synthetic type. That is, for example, when a synthetic type is specified for a particular component, the concrete type with the shortest execution time is chosen.

With each static adaptation iteration, RESAS gains more information about the software and hardware. It is therefore in a position to suggest modifications to the decisions made in the previous iteration. For example, it may be found to be desirable for the configuration algorithms to choose, for a particular component, not the fastest concrete type, but rather a concrete type which has better reliability characteristics. In the RESAS prototype, there are three choice points during the program configuration generation process:

- Choice of concrete type for an object (if the specified type of the object is synthetic).
- Choice of MaxTries for a basic object. (In the event of a failure, the maximum number of tries an object may make before giving up. See Section 3.5.)
- Choice of a processor to which a basic object is assigned.
For the static adaptation cycle, processor assignment is done by re-executing the processor assignment algorithm described in Chapter 7. Choice of a concrete type and MaxTries is done on an object-by-object basis, however. For each iteration of the static cycle, such choices must be communicated to the object instantiation algorithms from the adaptation choice algorithms operating on data from the previous iteration. This is done via hint relationships. Examples of these hint relationships are shown below:

**RELATIONSHIP**  
Name: TYPE-HINT  
Tuples:  
<table>
<thead>
<tr>
<th>object-instance</th>
<th>type</th>
<th>iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/DA</td>
<td>DupDataAcquisition</td>
<td>1</td>
</tr>
<tr>
<td>S/MP</td>
<td>DupMotionPlanning</td>
<td>2</td>
</tr>
</tbody>
</table>

**RELATIONSHIP**  
Name: MAXTRIES-HINT  
Tuples:  
<table>
<thead>
<tr>
<th>object-instance</th>
<th>max-tries</th>
<th>iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/IJ</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>S/FT</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Each iteration of the static algorithm uses, as a base, the configuration generated during the previous adaptation cycle. Therefore, there needs to be some way of dealing with old hints.

The solution used in the RESAS prototype is to make new Type hints remove any tuples in the MAXTRIES-HINT and TYPE-HINT
relationships which relate to objects in the subtree whose root is the target of the new Type hint. Since only basic objects may be assigned retries, there is no hierarchy for the MAXTRIES-HINT relationship. New MaxTries hints for an object cause the removal of any old hints (both MAXTRIES- and TYPE-) for the object.

8.4 Implementation of the Static Cycle

The static cycle works as follows: Assume that the current configuration is loaded onto the target hardware and executed. Sensors embedded in the target software report data to the DMS. The data of interest depends on the adaptation algorithms. Typically, these data may include the actual cpu time used by each activation, the processor loads and failure data, including missed deadlines and actual activation failures. As will be seen in Chapter 10, the overhead involved with these sensors is generally not significant.

At a certain point, either time-driven or event-driven, the adaptation controller begins an adaptation cycle. The cycle is run periodically in the tests done in this dissertation (see Chapter 10). The cycle may also be initiated by sensor data, such as an accumulation of failures or missed deadlines, or a hard processor or memory fault.
The AC first turns off the sensor system, so that the data in the DMS is frozen during adaptation algorithm execution. Next, it runs analysis algorithms to generate statistics such as average failure rates and processor loads.

Next, hints are generated for the TYPE-HINT and MAXTRIES-HINT relationships. These choices may be partially checked for feasibility in some cases (e.g. when the execution time of the replacing type is not significantly different from that of the replaced type). However, it is not always possible to do this without inordinately restricting the processor assignment algorithm, which does a complete reassignment at each iteration. Therefore, some form of backup mechanism or escape for human intervention is necessary.

The current prototype works as follows. The adaptation choice algorithm generates a single hint and attempts to regenerate the configuration. If it is unsuccessful (e.g. the objects will not fit onto the processors), the previous hint is removed and another hint is generated. If no successful hint can be found, the AC requests help from a human operator, who may then delete hints from or add hints to the hint relationships, in effect, either backtracking to a previous configuration (if they are deleted in reverse chronological order), or making a discontinuous jump to a new configuration.
8.5 Adaptation Choice Algorithms

Step 5 in the iterative static cycle in Section 8.2 refers to the use of adaptation choice algorithms. These algorithms use the data from one iteration of the static cycle and produce the hints described in the previous section. The role of such algorithms is to analyze the data, find weak spots either in real-time performance or reliability, and suggest hints which produce benefit at an acceptable cost.

Consider the following sample adaptation choice algorithm. This algorithm is designed to find basic objects which may benefit from a more reliable version (in particular, a duplicated version) or from an increase in MaxTries.
Begin;

Find the basic object instance BO with

\[ \text{MAX}[ \sum A.\text{service} \times A.\text{failures}] \]

where the sum is taken over all activations A of BO.

If \( \text{MaxTries} \) may be increased for this BO

Then HINT: set \( \text{MaxTries} := \text{MaxTries} + 1 \);
Regenerate the configuration;

If the regeneration was not successful and there is a duplicated type for this BO

Then HINT: set BO type := Duplicated;
Regenerate the configuration;

If the regeneration was not successful

Then ask the programmer to modify the HINT relationships;

End;

The choice of duplication or retry depends on the cost/benefit ratio and on whether both are even possible. If the entire configuration is redone at each iteration (as is the case in the prototype), the adaptation enactment cost is similar for both choices (This will be addressed in Chapter 10). Both duplication and retry, assuming independent failure probabilities for all executions of an activation, should decrease the failure rate considerably (proportional to the probability of two failures
of BO in a single application execution — assumed to be quite small in
the independent, transient fault model). Retry requires (potentially) more
processor power than a single BO execution (twice if MaxTries=2) on
the processor to which BO is assigned, possibly more than is available.
Duplication requires more than twice the processor power of a single BO
execution (forks, 2 copies of BO and joins) but it may be spread across
multiple processors.

The choice between duplication and retry is affected by the
"shape" of the software and hardware. If the application program is
tightly constrained by the deadlines along a critical path, then retry may
not be possible for objects on that path. However, if sufficient hardware
parallelism is available, duplication may be used for those objects, since
the two copies of each object may run in parallel. Conversely, if there is
sufficient room along the critical path, then retry may be more attrac­
tive, since it uses less processing power (i.e. no forks or joins) even
during failures, and uses no extra processing power in the absence of
failures. The algorithm shown above chooses retry if it is possible, and
if not, tries duplication. Some results of the use of this algorithm on a
sample program are given in Chapter 10.

If it is likely that an adaptation can be performed on one of the
several worst objects (in terms of service×failures), it may be cost
effective to search \(O(N)\) the set of objects for the worst one, try to adapt it and, if not successful, search for the second-worst object, and so on. If successful adaptation of the worst objects is not likely, it may be better to sort all of the objects first \(O(N \times \log N)\), and attempt adaptations on them in sorted order. Each attempted adaptation involves regenerating the configuration, with time complexity \(O(N \cdot M + N^2)\) (see Chapter 7).
Chapter 9
Dynamic Adaptation

9.1 Introduction

Dynamic adaptation is similar to iterative static adaptation. Many of the issues and mechanisms covered in Chapter 8 apply to both static and dynamic adaptations. Like the static cycle, the dynamic cycle starts with an "initial" configuration, which may be the result of the initial configuration generation (ICG) process or of previous static or dynamic cycles.

The initial configuration is executed on the actual hardware with either seeded or natural failure processes. Using these data, the dynamic adaptation algorithms are rerun, resulting in a new configuration which may be executed, and so on.
This dynamic adaptation cycle may be embedded into the executing system allowing the system to adapt to changing environmental conditions. The following sections describe the dynamic adaptation cycle and the mechanisms which implement it.

9.2 The Dynamic Cycle

The dynamic adaptation cycle consists of the following steps:
1. Load an "initial" configuration.
2. Initialize the appropriate attributes.
3. Execute the configuration and collect data.
4. After some time, stop the monitoring process and compile statistics.
5. Execute the dynamic adaptation choice algorithms (similar to the static algorithms), choosing particular changes to make to the current configuration.
6. Perform the changes using adaptation transactions (see Section 5.3.1).
7. Go to step 2.

As with iterative static adaptation, the point at which step 4 is executed depends on the nature of the application and on the failure rates and assumptions.
9.3 Adaptation Choice Algorithms

The adaptation choice algorithms use the data from one iteration of the dynamic cycle and produce adaptation decisions. The role of such algorithms is to analyze the data, find weak spots either in real-time performance or reliability, and suggest adaptations which produce benefit at an acceptable cost.

Because the adaptation choice and enactment take place on-the-fly, it is difficult to back up or request human intervention if an adaptation is found to be impossible. Therefore, the feasibility of an adaptation must be known (with a high degree of certainty) before it is attempted. This may affect the type of algorithms chosen for dynamic adaptations.

Consider the sample adaptation choice algorithm below. This algorithm is designed to find basic objects which may benefit from an increase in MaxTries or from reassignment to a more reliable processor.
Begin;

Find the basic object instance BO with

$$\text{MAX} \left[ \Sigma [ A.\text{service} \times A.\text{failures} ] \right]$$

where A is an activation associated with BO

If there is room on BO's processor to allow BO's MaxTries to be increased
Then set MaxTries := MaxTries+1;

Else If there is room on any other processor to allow BO's MaxTries to be increased
Then reassign BO to that processor and set MaxTries := MaxTries+1;

Else If a processor with fewer failures has room for BO
Then reassign BO to the best available processor
Else remove BO from contention for MAX above and try another BO;

End;

The choice of retry or reassigning BO depends on the cost/benefit ratio and on whether both are even possible. Reassigning BO to another processor should decrease the failure rate of BO by the same ratio as that of the overall failure rates of the old and new processors. It requires no more processor power, however, but only redistributes the load. Retry, assuming independent failure probabilities for all executions of an activation, should decrease the failure rate considerably (proportional to the probability of two failures of BO in a single
application execution — assumed to be quite small in the independent, transient fault model). Retry requires (potentially) more processor power than a single BO execution (twice if MaxTries=2) on the processor to which BO is assigned, possibly more than is available, but the potential benefit is greater than that of reassigning BO.

In terms of adaptation enactment cost, retry may be less costly, if not too many scheduling variables (starttimes and deadlines) need to be modified. For reassignment, the cost of moving BO to another processor depends on the number of other objects connected with it and the cost of reassigning schedules.

As with the static algorithm described in Chapter 8, the cost effectiveness of pre-sorting the objects depends on the likelihood of successfully performing an adaptation on one of the worse objects. The worst case time complexity of this algorithm is $O(N \times M)$, for $N$ objects and $M$ processors.

The results of tests of this algorithm on a sample program are discussed in Chapter 10.
Chapter 10
Tests of the RESAS Prototype

10.1 Introduction

This chapter describes the tests undertaken to gain an understanding of the capabilities of the RESAS prototype. It is beyond the scope of this dissertation to completely test all of the algorithms presented in the previous chapters on a large number of test programs. The goal of the experiments described below was to examine the strengths and weaknesses of the prototype mechanisms and to examine the results of the application of RESAS to a sample target program. Validation of the sample adaptation algorithms was not a goal of these experiments. Such algorithm testing requires a large number of experiments, using a variety of software and hardware configurations. This usually requires a high-speed simulation of the target system, which is not a part of the prototype.
10.2 Test Program

The Inverse Plant Plus Jacobian Control (IPPJC) program presented in Chapter 1 was used as a test program for the prototype RESAS mechanisms and sample algorithms described in previous chapters. It is shown again in Figure 21. Numbers on the invocation links represent the number of bytes in the invocation messages. The execution times of the software objects are also shown. (Assume, for the time being, that all of the objects are basic objects.)

This test program is initiated by an invocation of the DataAcquisition object (assumed to occur periodically) and produces one result through the OutputControl object. Thus, the program has one conceptual START object attached to DataAcquisition and one conceptual END object (with service value 1.0) attached to OutputControl. The objects (minus START and END) may be considered to be encapsulated in a SYSTEM object with one input interface and one output interface. The program has no built-in redundancy. Each object requires all input interfaces to be invoked before it may execute. The failure of any object causes the entire program to fail (and therefore, each object has service value 1). Each object has only one associated activation.
Figure 21: The Inverse Plant Plus Jacobian Control Program
10.3 RESAS Mechanism Tests

To gain an understanding of the capabilities of the prototype, measurements of the capacities and overheads of the object-based operating system extension, monitoring mechanisms and adaptation enactment mechanisms were performed.

10.3.1 Object Invocations

Object invocation latency on the target hardware averaged \((6.9 + 0.0018 \times N)\) milliseconds for \((N\text{-byte})\) invocations between objects on the same processor, and \((5.2 - 0.0027 \times N)\) milliseconds for objects on different processors. This is the elapsed time between the initiation of the Send operation on the invoker object and the entry into the application code of the invoked object. The lower constant term for inter-processor invocation is due to the fact that "clean-up" code executed by the invoker object after sending the invocation may be run in parallel with "start-up" code executed by the invoked object.

Object scheduling is complicated by the fact that processors are not capable of interrupting one another. Therefore, inter-processor invocations may not be recognized by the destination processor until the next clock interrupt (currently at 10ms intervals), although the time is generally
much shorter. These results show that the current prototype is most appropriate for programs with "large" objects, for which a 10ms variation in invocation times does not dominate object execution times. For this reason, the object execution times for the test program were multiplied by a factor of five during the tests.

10.3.2 Monitoring

Performance of the RESAS monitoring mechanism may be divided into two categories:

- Overhead of the Sense operation on the application objects.
- Overall throughput of the monitoring mechanism.

The overhead of the Sense operation on the application objects is approximately 750μs. This is the time it takes to insert a \((time, object, activation, attribute, value)\) message into a special queue in the shared memory of the multiprocessor.

Messages placed into the queue are formatted by the object on the processor P0 connected to the Sun workstation and passed, via a 9600 baud serial line, to the Sun. The monitoring code on the Sun processes the messages and places the values into the DMS. The serial connection between P0 and the Sun, and the Lisp DMS and MON code on the Sun impact the performance of the monitoring system. While the formatting
code on P0 can process approximately 62.5 messages per second (16ms/msg), the overall throughput of the monitoring system was only approximately 13.3 messages per second (75ms/msg). A typical sensor message, translated into approximately 25 ascii characters and sent to the Sun DMS, requires approximately 16ms for formatting on P0, approximately 35ms for transmission to the Sun, and approximately 24ms for insertion into the Sun DMS. The serial connection is clearly the bottleneck in the current prototype. This did not affect the results of the tests described in this chapter, but would be a factor for larger target programs.

10.3.3 Adaptation Enactment

In the dynamic adaptation mode, adaptations are enacted using the BeginAdaptationTransaction, XPutAtt and EndAdaptationTransaction operations described in Chapter 5.

The BeginAdaptationTransaction operation does not cause any overhead on the target program. It simply records the transaction id and transaction timeout values.

The XPutAtt operation passes a message from the DMS on the Sun, through P0, to the appropriate target object. Total throughput is approximately 16 XPutAtt operations per second (62ms/operation). As
with monitoring, the main bottleneck is the 9600 baud serial connection.
XPutAtt operations from P0 to the target object may be done at a rate of approximately 385 per second (2.6ms/operation).

The adaptation code of the target object receives an XPutAtt message through input interface I0, records the new value in a local table (but does not yet update the actual variable!) and sends an acknowledgment back to P0. The overhead on the target object is approximately 240\mu s. (An additional overhead of approximately 600\mu s is incurred during process scheduling and descheduling. This is true for all invocations.)

The EndAdaptationTransaction operation initiates Phase 2 of the adaptation transaction. During this phase, each object examines the XPutAtt messages logged in its local table and, if the timeout has not been passed, updates the actual attributes. The overhead on the target object of this action is approximately a constant 150\mu s plus 260\mu s per XPutAtt message. In the special case of copying the state of one object to another during object replacement, the overhead is approximately 1.8\mu s per byte if the source and destination objects are on the same processor and approximately 2.7\mu s if they are on different processors.
The adaptation transaction mechanism did not interfere with the normal flow of the target program in the tests described below. Normal adaptations changed only two or three attributes per object and object states were quite small (several bytes).

10.4 Fault Generator Tests

The underlying assumption about processor faults for the algorithms used in this dissertation is that the probability of a processor fault affecting an object is proportional to the object’s execution time.

Faults are simulated in the following way. Fault rates for the target processors (in faults per hour) are sent to a special “fault process” on P0. The fault process operates in an infinite cycle. Based on the fault rates, the fault process randomly chooses a processor. It then chooses a time at which to activate the fault. This time is chosen so that the average fault inter-arrival time corresponds to the total of the processors’ fault rates. Fault inter-arrival times (in milliseconds) are calculated by the formula:
FaultTime = CurrentTime + ApplicationPeriod +
X MOD [2 × (1/TotalFaultRate - ApplicationPeriod)]

where:

X is a random positive integer
TotalFaultRate is the sum of the processors' fault rates (in faults per millisecond)
ApplicationCycleTime is the period with which the START object invokes the target program

This results in fault inter-arrival times uniformly distributed in an interval which starts at ApplicationPeriod. Therefore, two faults will never appear within the same application cycle. This is necessary to approximate the "fault independence" assumption during tests with accelerated fault rates. That is, in actual target systems, fault inter-arrival times would be on the order of many hours. In these systems, independent faults would usually not occur within a single application cycle (on the order of several seconds). To approximate this in the tests, only single faults are generated.

Figure 22 shows the relationship between object execution time and object failures (normalized by processor fault rates) for several of the tests. There is a strong positive correlation between them and, as expected, a least squares linear approximation has a y-intercept near zero.
Figure 22: Object Execution Time vs. Normalized Object Failures

10.5 The Reliability Measure

The overall measure of target system reliability used for these tests is average service lost per fault. This measure describes the expected loss of service due to some randomly occurring processor fault. It is computed by:

\[
y = -0.0163 + 7.864 \times 10^{-4}x \quad R = 0.98
\]
\[ \text{SUM} [ X.\text{service} \times X.\text{failures} ] / \text{TotalProcessorFaults} \]

for all objects \( X \)

Processor faults occurring during periods when the processor is idle do not cause object failures. However, TotalProcessorFaults includes these faults, as well as faults which do cause object failures.

In a test with normal (non-accelerated) fault rates, missed END deadlines may be treated as faults, with the corresponding loss of service. Because the fault rates in these tests are highly accelerated, but the target program is running at approximately normal speed, missed deadlines in these tests are weighted with a \text{"very large\textquotedbl"} multiplier, and any target system configuration which exhibits missed END deadlines is considered unacceptable.

10.6 Iterative Static Adaptation Tests

The target system used for the tests consisted of the IPPJC test program and three processors (P3, P4 and P5) on the multiprocessor (in addition to P0). More processors were available, but the test program was not large enough to warrant their use.
The application cycle time and overall deadline for the test program was 1500ms. Fault rates of 500, 700 and 800 faults per hour were seeded for processors P3, P4 and P5 respectively, for a total fault rate of 2000 faults per hour. Tests ran for 15 minutes, allowing approximately 600 application cycles and approximately 500 processor faults.

10.6.1 Processor Assignment

In this test, only the static processor assignment algorithm of Section 7.7 was tested. For the initial configuration, all processor fault rates were assumed to be zero. This resulted in an assignment in which the objects were spread across the three processors:

RELATIONSHIP
Name: PROCESSOR-ASSIGNMENT
Tuples:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5</td>
<td>S/DA</td>
</tr>
<tr>
<td>P3</td>
<td>S/MP</td>
</tr>
<tr>
<td>P5</td>
<td>S/JC</td>
</tr>
<tr>
<td>P4</td>
<td>S/DK</td>
</tr>
<tr>
<td>P3</td>
<td>S/FT</td>
</tr>
<tr>
<td>P4</td>
<td>S/OL</td>
</tr>
<tr>
<td>P3</td>
<td>S/IJ</td>
</tr>
<tr>
<td>P4</td>
<td>S/JA</td>
</tr>
<tr>
<td>P5</td>
<td>S/ID</td>
</tr>
<tr>
<td>P4</td>
<td>S/IL</td>
</tr>
<tr>
<td>P5</td>
<td>S/OC</td>
</tr>
</tbody>
</table>

The configuration was compiled, linked and loaded onto the multiprocessor. Results of the test were:
Application Cycles: 601

Missed END Deadlines: 0

Total Processor Faults: 496
   P3: 141
   P4: 177
   P5: 178

<table>
<thead>
<tr>
<th>Object</th>
<th>Failures</th>
<th>ServiceLost</th>
<th>ExecTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/ID</td>
<td>50</td>
<td>50.0</td>
<td>398 ms</td>
</tr>
<tr>
<td>S/JA</td>
<td>37</td>
<td>37.0</td>
<td>254</td>
</tr>
<tr>
<td>S/JC</td>
<td>19</td>
<td>19.0</td>
<td>134</td>
</tr>
<tr>
<td>S/MP</td>
<td>13</td>
<td>13.0</td>
<td>145</td>
</tr>
<tr>
<td>S/DK</td>
<td>13</td>
<td>13.0</td>
<td>114</td>
</tr>
<tr>
<td>S/IJ</td>
<td>8</td>
<td>8.0</td>
<td>89</td>
</tr>
<tr>
<td>S/DA</td>
<td>7</td>
<td>7.0</td>
<td>59</td>
</tr>
<tr>
<td>S/FT</td>
<td>3</td>
<td>3.0</td>
<td>19</td>
</tr>
<tr>
<td>S/OL</td>
<td>1</td>
<td>1.0</td>
<td>34</td>
</tr>
</tbody>
</table>

(all others 0)

Average Service Lost Per Fault: 0.3044

As expected, S/ID, which has the greatest execution time and was assigned to the most faulty processor, experienced the most failures. (For the objects in these tests, ServiceLost is the same as Failures, since each object has ServiceValue=1.)

In the second iteration, the static assignment algorithm moved objects from P5 to the other processors:
RELATIONSHIP
Name: PROCESSOR-ASSIGNMENT
Tuples:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>S/DA</td>
</tr>
<tr>
<td>P4</td>
<td>S/MP</td>
</tr>
<tr>
<td>P3</td>
<td>S/JC</td>
</tr>
<tr>
<td>P4</td>
<td>S/DK</td>
</tr>
<tr>
<td>P4</td>
<td>S/FT</td>
</tr>
<tr>
<td>P4</td>
<td>S/OL</td>
</tr>
<tr>
<td>P4</td>
<td>S/IJ</td>
</tr>
<tr>
<td>P3</td>
<td>S/JA</td>
</tr>
<tr>
<td>P3</td>
<td>S/ID</td>
</tr>
<tr>
<td>P3</td>
<td>S/IL</td>
</tr>
<tr>
<td>P3</td>
<td>S/OC</td>
</tr>
</tbody>
</table>

This resulted in an Average Service Lost Per Fault of 0.2390.

Figure 23 shows the results of subsequent cycles. Despite the fact that P5 was seeded with a higher number of faults than P4, due to the random nature of the fault generation algorithm, the number of faults exhibited by P5 occasionally dipped slightly below the number of faults exhibited by P4. In such circumstances, the objects assigned to P4 were reassigned to P5, and on the following cycle, were generally reassigned back to P4. This cycling of object assignments between P4 and P5 was due to the short-term nature of the algorithm. That is, only fault information from the immediately previous cycle is used by the assignment algorithm. An appropriately weighted fault history from all previous cycles would even out this instability.
Despite this instability, the average service lost per fault for all nine adaptation cycles averaged 0.2453, an improvement of approximately 20%\(^9\) over the default assignment's 0.3044.

\(^9\)These comparisons should be read with the understanding that the random nature of the processor faults causes variations of several hundredths of a point in service lost during successive tests of identical configurations.
10.6.2 Processor Assignment and MaxTries

In this test, the processor assignment algorithm was combined with the hint algorithm of Section 8.3. Each object was initially assigned MaxTries=1 and the AC was allowed to increase each object's MaxTries to two, effectively allowing one retry if a failure occurred on the first try. No concrete type choice was allowed.

The initial configuration was exactly as in the assignment test, with results:

Application Cycles: 601

Missed END Deadlines: 0

Total Processor Faults: 494
   P3: 133
   P4: 174
   P5: 187

<table>
<thead>
<tr>
<th>Object</th>
<th>ServiceLost</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/ID</td>
<td>56.0</td>
</tr>
<tr>
<td>S/JA</td>
<td>26.0</td>
</tr>
<tr>
<td>S/JC</td>
<td>24.0</td>
</tr>
<tr>
<td>S/MP</td>
<td>13.0</td>
</tr>
<tr>
<td>S/DK</td>
<td>9.0</td>
</tr>
<tr>
<td>S/IJ</td>
<td>7.0</td>
</tr>
<tr>
<td>S/OL</td>
<td>5.0</td>
</tr>
<tr>
<td>S/DA</td>
<td>4.0</td>
</tr>
<tr>
<td>S/IL</td>
<td>2.0</td>
</tr>
<tr>
<td>S/FT</td>
<td>1.0</td>
</tr>
<tr>
<td>(all others 0)</td>
<td></td>
</tr>
</tbody>
</table>

Average Service Lost Per Fault: 0.2976
In the second cycle, MaxTries was set to 2 for S/ID, and the objects were reassigned:

**RELATIONSHIP**

**Name:** PROCESSOR-ASSIGNMENT

**Tuples:**

<table>
<thead>
<tr>
<th>Processor</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>S/DA</td>
</tr>
<tr>
<td>P4</td>
<td>S/MP</td>
</tr>
<tr>
<td>P3</td>
<td>S/JC</td>
</tr>
<tr>
<td>P4</td>
<td>S/DK</td>
</tr>
<tr>
<td>P4</td>
<td>S/FT</td>
</tr>
<tr>
<td>P4</td>
<td>S/OL</td>
</tr>
<tr>
<td>P4</td>
<td>S/IJ</td>
</tr>
<tr>
<td>P3</td>
<td>S/JA</td>
</tr>
<tr>
<td>P3</td>
<td>S/ID</td>
</tr>
<tr>
<td>P3</td>
<td>S/IL</td>
</tr>
<tr>
<td>P3</td>
<td>S/OC</td>
</tr>
</tbody>
</table>

This resulted in an Average Service Lost Per Fault of 0.1840.

Figure 23 shows the results of subsequent cycles. The adaptations chosen were:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Adaptation</th>
<th>ServiceLostPerFault</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>default</td>
<td>0.2976</td>
</tr>
<tr>
<td>2</td>
<td>S/ID MaxTries=2</td>
<td>0.1840</td>
</tr>
<tr>
<td>3</td>
<td>S/IJ MaxTries=2</td>
<td>0.1554</td>
</tr>
<tr>
<td>4</td>
<td>S/DK MaxTries=2</td>
<td>0.1142</td>
</tr>
<tr>
<td>5</td>
<td>S/OL MaxTries=2</td>
<td>0.1202</td>
</tr>
<tr>
<td>6</td>
<td>S/IL MaxTries=2</td>
<td>0.1225</td>
</tr>
</tbody>
</table>

After cycle 6, no more MaxTries adaptations were possible. The average service lost per fault for cycle 6 was an improvement of approximately 60% over the default assignment.
10.6.3 Processor Assignment and Duplication

In this test, the processor assignment algorithm was again combined with the hint algorithm of Section 8.3. Each object was provided with a basic type and also a composite "duplicated" type as shown in Figure 11. MaxTries was fixed at one for all objects.

The initial configuration was exactly as in the assignment test, with results:

**Application Cycles:** 601

**Missed END Deadlines:** 0

**Total Processor Faults:** 497
- P3: 140
- P4: 162
- P5: 195

<table>
<thead>
<tr>
<th>Object</th>
<th>ServiceLost</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/ID</td>
<td>44.0</td>
</tr>
<tr>
<td>S/JA</td>
<td>29.0</td>
</tr>
<tr>
<td>S/JC</td>
<td>25.0</td>
</tr>
<tr>
<td>S/MP</td>
<td>12.0</td>
</tr>
<tr>
<td>S/DK</td>
<td>12.0</td>
</tr>
<tr>
<td>S/IJ</td>
<td>10.0</td>
</tr>
<tr>
<td>S/DA</td>
<td>5.0</td>
</tr>
<tr>
<td>S/FT</td>
<td>3.0</td>
</tr>
<tr>
<td>S/OL</td>
<td>3.0</td>
</tr>
<tr>
<td>S/OC</td>
<td>1.0</td>
</tr>
<tr>
<td>(all others 0)</td>
<td></td>
</tr>
</tbody>
</table>

**Average Service Lost Per Fault:** 0.2898

In the second cycle, the duplicated type was chosen for S/ID, and the objects were reassigned:
RELATIONSHIP
Name: PROCESSOR-ASSIGNMENT
Tuples:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>S/DA</td>
</tr>
<tr>
<td>P4</td>
<td>S/MP</td>
</tr>
<tr>
<td>P3</td>
<td>S/ID/P2</td>
</tr>
<tr>
<td>P3</td>
<td>S/JC</td>
</tr>
<tr>
<td>P5</td>
<td>S/DK</td>
</tr>
<tr>
<td>P5</td>
<td>S/FT</td>
</tr>
<tr>
<td>P5</td>
<td>S/OL</td>
</tr>
<tr>
<td>P5</td>
<td>S/IJ</td>
</tr>
<tr>
<td>P3</td>
<td>S/JA</td>
</tr>
<tr>
<td>P3</td>
<td>S/ID/F1</td>
</tr>
<tr>
<td>P3</td>
<td>S/ID/P3</td>
</tr>
<tr>
<td>P4</td>
<td>S/ID/C1</td>
</tr>
<tr>
<td>P3</td>
<td>S/ID/C2</td>
</tr>
<tr>
<td>P3</td>
<td>S/ID/J1</td>
</tr>
<tr>
<td>P3</td>
<td>S/IL</td>
</tr>
<tr>
<td>P3</td>
<td>S/OC</td>
</tr>
</tbody>
</table>

This resulted in an Average Service Lost Per Fault of 0.1964.

Figure 23 shows the results of subsequent cycles. The adaptations chosen were:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Adaptation</th>
<th>ServiceLostPerFault</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>default</td>
<td>0.2898</td>
</tr>
<tr>
<td>2</td>
<td>S/ID Duplicated</td>
<td>0.1964</td>
</tr>
<tr>
<td>3</td>
<td>S/JA Duplicated</td>
<td>0.1526</td>
</tr>
<tr>
<td>4</td>
<td>S/MP Duplicated</td>
<td>0.1308</td>
</tr>
<tr>
<td>5</td>
<td>S/DK Duplicated</td>
<td>0.1158</td>
</tr>
<tr>
<td>6</td>
<td>S/IJ Duplicated</td>
<td>0.0541 (Missed Deadlines)</td>
</tr>
</tbody>
</table>

During cycle 6, five END deadlines were missed. Therefore, the configuration was rolled back to that in cycle 5. No other adaptations
were possible. The average service lost per fault for cycle 5 was an improvement of approximately 60% over the default assignment.

10.6.4 Processor Assignment, MaxTries and Duplication

In this test, the processor assignment algorithm was again combined with the hint algorithm of Section 8.3. Each object was provided with a basic type and also a composite “duplicated” type. MaxTries was initially set to one for all basic objects, but was allowed to be increased to two.

The initial configuration was exactly as in the assignment test, with results:
Application Cycles: 600

Missed END Deadlines: 0

Total Processor Faults: 499
P3: 136
P4: 182
P5: 181

<table>
<thead>
<tr>
<th>Object</th>
<th>ServiceLost</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/ID</td>
<td>57.0</td>
</tr>
<tr>
<td>S/JA</td>
<td>34.0</td>
</tr>
<tr>
<td>S/DK</td>
<td>16.0</td>
</tr>
<tr>
<td>S/JC</td>
<td>12.0</td>
</tr>
<tr>
<td>S/MP</td>
<td>8.0</td>
</tr>
<tr>
<td>S/IJ</td>
<td>6.0</td>
</tr>
<tr>
<td>S/DA</td>
<td>6.0</td>
</tr>
<tr>
<td>S/OL</td>
<td>3.0</td>
</tr>
<tr>
<td>S/FT</td>
<td>2.0</td>
</tr>
<tr>
<td>(all others 0)</td>
<td></td>
</tr>
</tbody>
</table>

Average Service Lost Per Fault: 0.2886

In the second cycle, MaxTries was set to 2 for S/ID, and the objects were reassigned:
RELATIONSHIP
Name: PROCESSOR-ASSIGNMENT
Tuples:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>S/DA</td>
</tr>
<tr>
<td>P5</td>
<td>S/MP</td>
</tr>
<tr>
<td>P3</td>
<td>S/JC</td>
</tr>
<tr>
<td>P5</td>
<td>S/DK</td>
</tr>
<tr>
<td>P5</td>
<td>S/FT</td>
</tr>
<tr>
<td>P5</td>
<td>S/OL</td>
</tr>
<tr>
<td>P5</td>
<td>S/IJ</td>
</tr>
<tr>
<td>P3</td>
<td>S/JA</td>
</tr>
<tr>
<td>P3</td>
<td>S/ID</td>
</tr>
<tr>
<td>P3</td>
<td>S/IL</td>
</tr>
<tr>
<td>P3</td>
<td>S/OC</td>
</tr>
</tbody>
</table>

This resulted in an Average Service Lost Per Fault of 0.2116.

Figure 23 shows the results of subsequent cycles. The adaptations chosen were:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Adaptation</th>
<th>ServiceLostPerFault</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>default</td>
<td>0.2886</td>
</tr>
<tr>
<td>2</td>
<td>S/ID MaxTries=2</td>
<td>0.2116</td>
</tr>
<tr>
<td>3</td>
<td>S/JA Duplicated</td>
<td>0.1403</td>
</tr>
<tr>
<td>4</td>
<td>S/MP Duplicated</td>
<td>0.1024</td>
</tr>
<tr>
<td>5</td>
<td>S/DK MaxTries=2</td>
<td>0.0998</td>
</tr>
<tr>
<td>6</td>
<td>S/IJ MaxTries=2</td>
<td>0.0663</td>
</tr>
<tr>
<td>7</td>
<td>S/OL MaxTries=2</td>
<td>0.0442</td>
</tr>
<tr>
<td>8</td>
<td>S/DA Duplicated</td>
<td>0.0221 (Missed Deadlines)</td>
</tr>
</tbody>
</table>

During cycle 8, 38 END deadlines were missed. Therefore, the configuration was rolled back to that in cycle 7. No other adaptations were possible. The average service lost per fault for cycle 7 was an improvement of approximately 85% over the default assignment.
10.6.5 Discussion of the Static Tests

The main conclusion which can be drawn from these tests is that the application of the RESAS algorithms and mechanisms did indeed produce increased reliability. As shown in the last static test, a mixture of techniques (e.g. duplication and retry), each with partial applicability, provides more reliability improvement than any single technique. Duplication is more useful in cases where there is available hardware parallelism, while retry is more appropriate if there is sufficient available time in the schedule (even on a uniprocessor). Furthermore, different fault models may require different techniques (e.g. triplication).

The missed END deadlines in several of the tests are the result of "calibration errors" in the algorithms. If the actual target system behaved in exactly the way it is assumed to behave (i.e. invocation times, object execution times, RESAS overheads), the deadlines would not be missed. In general, this problem may be avoided by using worst-case bounds on target system activities, and by providing some amount of spare resources for unanticipated extreme cases. It is reasonable to expect normal processor loads to be kept, for example, below 80%. (The Space Shuttle computer system is normally loaded about 75% [19].)

Spare resources are also helpful, although not absolutely necessary, for taking advantage of adaptability. Sparser processor schedules allow
more object duplication or reassignment, and heuristic adaptation algorithms (e.g. processor assignment algorithms) produce better results when loads are not too high.

The processor assignment algorithm does decrease service lost per fault. In these tests, the amount of improvement is not dramatic, due to the processor fault rates. The worst possible assignment would be to have all objects on P5, with a fault rate of 800 per hour. The best possible assignment would be to have all objects on P3, with a fault rate of 500 per hour. Thus, the largest possible improvement would be approximately 37.5%. In the tests, the initial assignment was somewhat better than the worst case, so improvement was even less dramatic. In actual target systems, one processor may be significantly more faulty than the others. In these cases, the maximum possible improvement is greater.

The static assignment algorithm did not, however, generate the best possible assignment. Generally, the objects were placed on both P3 and P4. This is due to the fact that the algorithm assigns all objects in one sweep, not using information from previous assignments, and must include worst case object start times in the decision. Thus, even though P4 was more faulty than P3, objects could occasionally be assigned later start times on P4. As will be seen in the next section, the dynamic
assignment algorithm (an incremental reassignment algorithm) did generate the best possible assignment.

While the Lisp implementation of the prototype is slow in comparison to more efficiency-conscious implementations which are possible (in Lisp or some other language), the performance of the prototype is still reasonably fast. Generation of a complete configuration for the 11-object test program (plus 33 spare objects distributed among the processors) requires approximately 50 seconds. This includes object instantiation, processor assignment and generation of the configuration files describing the compilation, linking and loading operations. This time is dominated by the time taken by the Intel development system to perform the compilation, linking and loading (approximately 6 minutes per processor). For static adaptations, such times are acceptable, since target system service need only be interrupted during the actual loading of the new configuration.
10.7 Dynamic Adaptation Tests

The target system used for these tests was the same as that used for the static tests.

10.7.1 Processor Assignment

In this test, only the dynamic processor assignment algorithm of Section 9.3 was tested. The initial configuration was the same as for the static tests. In addition to the basic objects "in the configuration", instances of each basic object type were placed on each processor, to be used as "shadow copies" when moving objects from processor to processor. The configuration was compiled, linked and loaded onto the multiprocessor. Results of the test were:
Application Cycles: 600

Missed END Deadlines: 0

Total Processor Faults: 501
P3: 124
P4: 167
P5: 210

<table>
<thead>
<tr>
<th>Object</th>
<th>Service Lost</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/ID</td>
<td>51.0</td>
</tr>
<tr>
<td>S/JA</td>
<td>20.0</td>
</tr>
<tr>
<td>S/JC</td>
<td>15.0</td>
</tr>
<tr>
<td>S/DA</td>
<td>13.0</td>
</tr>
<tr>
<td>S/MP</td>
<td>11.0</td>
</tr>
<tr>
<td>S/DK</td>
<td>11.0</td>
</tr>
<tr>
<td>S/OL</td>
<td>8.0</td>
</tr>
<tr>
<td>S/IJ</td>
<td>4.0</td>
</tr>
<tr>
<td>S/FT</td>
<td>2.0</td>
</tr>
<tr>
<td>S/IL</td>
<td>2.0</td>
</tr>
</tbody>
</table>

(all others 0)

Average Service Lost Per Fault: 0.2735

Unlike the static processor assignment algorithm, which reassigns each object during each adaptation cycle, the dynamic algorithm moves only one object per adaptation cycle.

Figure 24 shows the results of subsequent cycles. The adaptations chosen were:
The result was that all objects were moved to P3, the least faulty processor, generating the best possible assignment. The average service
lost per fault for cycle 9 was an improvement of approximately 15% over the default assignment.

10.7.2 MaxTries

In this test, only the dynamic MaxTries choice algorithm of Section 9.3 was tested. Object reassignment was not allowed. The initial configuration was the same as for the previous tests.

Application Cycles: 600

Missed END Deadlines: 0

Total Processor Faults: 497
P3: 123
P4: 181
P5: 193

<table>
<thead>
<tr>
<th>Object</th>
<th>ServiceLost</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/ID</td>
<td>50.0</td>
</tr>
<tr>
<td>S/JA</td>
<td>32.0</td>
</tr>
<tr>
<td>S/JC</td>
<td>23.0</td>
</tr>
<tr>
<td>S/DK</td>
<td>19.0</td>
</tr>
<tr>
<td>S/MP</td>
<td>18.0</td>
</tr>
<tr>
<td>S/DA</td>
<td>8.0</td>
</tr>
<tr>
<td>S/IJ</td>
<td>5.0</td>
</tr>
<tr>
<td>S/OL</td>
<td>4.0</td>
</tr>
<tr>
<td>S/FT</td>
<td>2.0</td>
</tr>
<tr>
<td>S/IL</td>
<td>2.0</td>
</tr>
</tbody>
</table>

(all others 0)

Average Service Lost Per Fault: 0.3280

Figure 24 shows the results of subsequent cycles. The adaptations chosen were:
<table>
<thead>
<tr>
<th>Cycle Adaptation</th>
<th>ServiceLostPerFault</th>
</tr>
</thead>
<tbody>
<tr>
<td>1    default</td>
<td>0.3280</td>
</tr>
<tr>
<td>2    S/ID MaxTries=2</td>
<td>0.1891</td>
</tr>
<tr>
<td>3    S/DK MaxTries=2</td>
<td>0.1796</td>
</tr>
<tr>
<td>4    S/IJ MaxTries=2</td>
<td>0.1567</td>
</tr>
<tr>
<td>5    S/DA MaxTries=2</td>
<td>0.1210</td>
</tr>
</tbody>
</table>

After cycle 5, no other adaptations were possible. The average service lost per fault for cycle 5 was an improvement of approximately 65% over the default assignment.

10.7.3 Processor Assignment and MaxTries

In this test, both dynamic MaxTries choice and dynamic processor assignment were allowed. The initial configuration was the same as for the previous tests. As before, shadow copies were placed on all processors.
Application Cycles: 600

Missed END Deadlines: 0

Total Processor Faults: 499
P3: 106
P4: 186
P5: 207

<table>
<thead>
<tr>
<th>Object</th>
<th>ServiceLost</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/ID</td>
<td>62.0</td>
</tr>
<tr>
<td>S/JA</td>
<td>27.0</td>
</tr>
<tr>
<td>S/JC</td>
<td>21.0</td>
</tr>
<tr>
<td>S/IJ</td>
<td>18.0</td>
</tr>
<tr>
<td>S/DK</td>
<td>16.0</td>
</tr>
<tr>
<td>S/MP</td>
<td>15.0</td>
</tr>
<tr>
<td>S/DA</td>
<td>12.0</td>
</tr>
<tr>
<td>S/OL</td>
<td>5.0</td>
</tr>
<tr>
<td>S/OC</td>
<td>3.0</td>
</tr>
<tr>
<td>S/FT</td>
<td>2.0</td>
</tr>
<tr>
<td>(all others 0)</td>
<td></td>
</tr>
</tbody>
</table>

Average Service Lost Per Fault: 0.3627

Figure 24 shows the results of subsequent cycles. The adaptations chosen were:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Adaptation</th>
<th>ServiceLost Per Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>default</td>
<td>0.3627</td>
</tr>
<tr>
<td>2</td>
<td>S/ID MaxTries=2</td>
<td>0.1940</td>
</tr>
<tr>
<td>3</td>
<td>S/DK MaxTries=2</td>
<td>0.1556</td>
</tr>
<tr>
<td>4</td>
<td>S/IJ MaxTries=2</td>
<td>0.1512</td>
</tr>
<tr>
<td>5</td>
<td>S/OL MaxTries=2</td>
<td>0.1647</td>
</tr>
<tr>
<td>6</td>
<td>S/DA MaxTries=2</td>
<td>0.1308</td>
</tr>
</tbody>
</table>

After cycle 6, no other adaptations were possible. Unfortunately, this test did not exercise the full algorithm. It was not possible to move
any objects, so this test was reduced to a second test of the MaxTries algorithm of the previous section. The average service lost per fault for cycle 6 was an improvement of approximately 65% over the default assignment.

10.7.4 Discussion of the Dynamic Tests

As with the static tests, the dynamic tests demonstrated increases in reliability of the target system. Many of the points raised in the discussion of the static tests apply to dynamic adaptations as well. Unlike the static cycle, however, the dynamic cycle is envisioned as being able to respond rapidly to conditions warranting adaptations. For the dynamic tests, total RESAS response time was on the order of 30 seconds, broken down as follows:

1. Collection of "final" data: approximately 5 seconds.
2. Generation of statistics: approximately 5 seconds.
3. Selection of an adaptation: 4 to 19 seconds.
4. Enactment of the adaptation: approximately 9 seconds.

Such response times are acceptable for some real-time environments and fault models, but for others a factor of 100 to 1000 speedup is necessary. Even this is not an unreasonable goal, however (e.g. by abandoning interpreted Lisp and 9600 baud communication!).
With dynamic adaptations, the cost of an adaptation is a factor in the selection of adaptations. While the sample dynamic algorithms did not address this issue, it is an important and interesting one. Many types of dynamic adaptations have costs which may be easily calculated. For example, the cost of moving an object from one processor to another is a function of the cost of moving the object state, reconnecting invocation links and possibly changing object start times and deadlines.

The dynamic processor assignment algorithm, unlike the static algorithm, resulted in the best possible assignment after several iterations. It would be possible, in the static cycle, to combine the static algorithm with several iterations of the dynamic algorithm, resulting in successive refinements of the assignment.

Many other algorithms are possible. It can be argued that since in the given fault model object failures depend on execution time, adapting two small objects has the same benefit as adapting one large object with the same total execution time. In fact, adapting smaller objects first may lead to more balanced processor schedules and less overall adaptation cost.

The general conclusion of these tests is that a system which supports many different techniques and algorithms is a valuable tool for increasing target system reliability.
Chapter 11
Conclusion

The RESAS prototype is an attempt to provide a uniform system for adapting real-time software to enhance its performance and reliability. A number of conclusions may be drawn from experience with the RESAS prototype.

11.1 Lessons Learned from the Prototype

The RESAS object model is designed to support the construction of large programs for which adaptations may be chosen and enacted with respect to real-time performance and reliability. The hierarchical construction of objects and the ease with which invocation connections may be modified have made adaptations easy to conceptualize and perform. The model is relatively easy to use and is appropriate for many real-time applications, but not all. It is not sufficiently general to handle
objects with multiple operations on the same encapsulated data, for example. Extensions to the model are straightforward. However, adaptation algorithms may become more complex, as other object interactions (e.g. the sharing of data between basic objects) are made possible.

The RESAS entity/relationship framework is a useful representation of the target system. In fact, during the algorithm tests, implementation inefficiencies in the prototype DMS occasionally prompted us to artificially bypass the ER framework, encoding "relationship" data in entities, for example. This invariably resulted in hidden bugs which cost a good deal of programmer time to track down. Returning to the ER model usually ended up being the most intelligent course of action. The ER operations provide a powerful, uniform (and, after some practice, easy-to-use) "language" for writing adaptation algorithms.

Action routines are provided primarily to maintain consistency between target system and its DMS representation, and to encode dependencies (or "relationships") between entity and relationship attributes in the DMS. Action routines are relatively effective at the former task. This is because there are not many different types of action routines in this category. A DMS PutAtt operation either is or is not forwarded to the target system, and a DMS GetAtt operation either does or does not query the target system. Action routines are less effective when used to
represent dependencies between entity and relationship attributes in the DMS. It is difficult to maintain an understanding of these "relationships", resulting in side effects which are unanticipated. While action routines are a useful implementation tool, a formal declarative method for describing these types of dependencies would be more appropriate.

The generation of views of the data simplifies algorithm writing. The "view-oriented" nature of the prototype implementation is efficient and powerful. However, care must be taken in maintaining the correctness of such views, since changing attribute values in one view may invalidate other views.

Adaptation transactions, as described in this dissertation, worked quite well. This transaction construct, which maintains consistency between the DMS and the target system during adaptation enactment, needs to be combined with a transaction construct within the DMS itself which constructs and executes adaptation transactions only if changes to the DMS can be successfully made. For example, it should be possible to start a transaction, make several modifications to the PROCESSOR-ASSIGNMENT relationship and, depending if a particular change is possible, either construct and execute an adaptation transaction to enact the changes or abort and roll back the DMS representation.
Preliminary tests with the adaptation algorithms described in this dissertation indicate that it is indeed possible to collect sufficient data from the target system, to produce adaptation decisions and to enact adaptations resulting in marked improvement in real-time performance and reliability. It would be relatively easy to encode many of the extant algorithms (e.g. processor assignment, object replication) and techniques (e.g. triplication) in terms of objects and within the entity/relationship framework. This is the real power of RESAS.

11.2 Larger Systems

The need for the mechanisms presented in this dissertation cannot be fully appreciated in the context of a 50-object program running on an eight-processor multiprocessor. In the case of a program with hundreds or thousands of objects running on a machine with tens or hundreds of processors, however, the need for the flexibility that a uniform software adaptation system provides is more apparent. Analysis of the performance of the RESAS prototype leads to some predictions of the effectiveness of a "second generation" RESAS on larger target systems.

The monitoring and adaptation enactment mechanisms did not impose undue overhead on the target system, and these overheads were
proportional to the amount of use of the mechanisms. Therefore, it would not be unreasonable to build these mechanisms into a runtime system, even if they were used infrequently. The ability to selectively "turn on" and "turn off" sensors would allow much larger target systems to be monitored without overloading the DMS. A more powerful monitoring system is described in [56]. Also, higher speed communication between the target system and the DMS would increase monitoring performance considerably. A streamlined DMS implementation residing on one of the processors of the current multiprocessor could sustain (estimated) rates on the order of 1000 sensor messages per second, a 75-fold increase over the current implementation.

The amount of data stored in the DMS for a program is not unreasonable. Approximately 100 to 200 bytes of data per object would be required in an efficient implementation of the DMS. Entities for 1000 objects could therefore be stored in a single 256K RAM.

The sample adaptation algorithms are generally of low time complexity (usually less than O(N^2) for N objects) and should therefore be usable for configurations which are somewhat larger than those used in this dissertation.
Thus, the overhead of the RESAS prototype is not unreasonable. For example, rough estimates indicate that such a system, more efficiently implemented and scaled up to handle 500 to 1000 objects on a 32-node hypercube machine might require one or two of the nodes to be dedicated to RESAS. This is not a large overhead, considering the benefits of flexibility, reliability and uniformity offered by RESAS. Similar estimates should hold for larger (e.g. 128-node) systems.

11.3 Future Research

This dissertation covers a wide range of topics. Many interesting topics have been touched upon and would benefit from further investigation.

The RESAS prototype data management system and adaptation controller are centralized. A second-generation RESAS should allow these mechanisms to be distributed. Distributed decision-making requires that the issue of hierarchical (or, in general, coupled) control systems be addressed with respect to real-time and reliability requirements.

Application-specific adaptations require a more formal and user-friendly method of describing adaptations.
The transfer of state from one object to another during adaptation transactions deserves more investigation. While it may make use of some current research [13], this problem must be approached in conjunction with the adaptation transaction mechanism, with special consideration for the real-time and reliability requirements of the problem domain.

The issue of error recovery needs to be addressed if RESAS is to be a practical tool for real-time systems. Other fault models and their associated algorithms deserve investigation.

All of these topics, and many others, may be addressed by extension of the current RESAS prototype. This is yet another indication of the power of such a uniform system.
List Of References


42. Barbara Liskov and Robert Scheifler. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs". *ACM Transactions on Programming Languages and Systems* (July 1983), 381-404.


