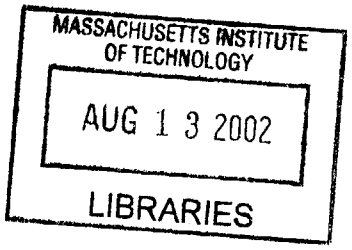
A Flight Software Development and Simulation Framework for Advanced Space Systems

by

John Patrick Enright

B.A.Sc., Engineering Science
University of Toronto, 1997

S.M, Aeronautics and Astronautics
Massachusetts Institute of Technology, 1999



SUBMITTED TO THE DEPARTMENT OF AERONAUTICS & ASTRONAUTICS
IN PARTIAL FULFILLMENT OF THE DEGREE OF

DOCTOR OF PHILOSOPHY

AT THE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUNE, 2002

Signature of Author.....
Department of Aeronautics and Astronautics
May 24, 2002

Certified by.....
Professor David Miller
Committee Chairman

Certified by.....
Professor Brian Williams
Committee Member

Certified by.....
Dr. Lorraine Fesq
Committee Member

Certified by.....
Dr. Raymond Sedwick
Committee Member

Accepted by.....
Professor Wallace E. Vander Velde
Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Studies

A Flight Software Development and Simulation Framework for Advanced Space Systems

by

JOHN ENRIGHT

Submitted to the Department of Aeronautics and Astronautics
on May 24, 2002 in Partial Fulfillment of the
Requirements for the Doctor of Philosophy
at the Massachusetts Institute of Technology

ABSTRACT

Distributed terrestrial computer systems employ middleware software to provide communications abstractions and reduce software interface complexity. Embedded applications are adopting the same approaches, but must make provisions to ensure that hard real-time temporal performance can be maintained. This thesis presents the development and validation of a middleware system tailored to spacecraft flight software development. Our middleware runs on the Generalized Flight Operations Processing Simulator (GFLOPS) and is called the GFLOPS Rapid Real-time Development Environment (GRRDE). GRRDE provides publish-subscribe communication services between software components. These services help to reduce the complexity of managing software interfaces. The hard real-time performance of these services has been verified with General Timed Automata modelling and extensive run-time testing. Several example applications illustrate the use of GRRDE to support advanced flight software development. Two technology-focused studies examine automatic code generation and autonomous fault protection within the GRRDE framework. A complex simulation of the TechSat 21 distributed space-based radar mission highlights the utility of the approach for large-scale applications.

Thesis Advisor:
Prof. David. W. Miller (chair)
Dept. of Aeronautics and Astronautics

ACKNOWLEDGMENTS

After much effort, it is all over. One day I might become a collector of doctoral degrees, but until that time, I must thank all of those people who made this momentous undertaking possible.

First of all I would like to take this opportunity to thank my advisor, Prof. David Miller. A man more committed to the success of his grad students I have never met. Your drive and dynamism is an inspiration to everyone around. And by the way, thanks for the coffee!

Prof. Brian Williams, thank you for the guidance about all things autonomous. You helped to spark my interest in the potential of smart spacecraft. I have found your feedback insightful.

Dr. Lorraine Fesq, I must certainly thank you for all that you've done. You're a great source of insight into the realities of flight software. Always ready to lend a hand, you constantly go the extra distance for your students.

Next, I would like to give my heartfelt thanks to Dr. Raymond Sedwick. Always ready to hash out a problem, always ready to lend a hand, always ready to go for coffee (Do you see a trend here?). I have to thank you for your invaluable help, guidance, and friendship.

To my readers, Prof. Kristina Lundqvist and Dr. Natasha Neogi, thank you for your time and feedback. This document is much better for it. I must also thank Prof. Nancy Lynch for all the help and suggestions you gave me on formal modelling.

Without a doubt the next thanks must go to Mr. Paul Bauer. Paul, you're an absolutely vital part of the lab. And another coffee connoisseur.

Ms. SharonLeah Brown, our fiscal officer, I thank you for all that you do for us in the lab. You keep the wheels of the lab turning. Ms. Margaret (Peggy) Edwards and more recently

Ms. Marilyn Good, thank you for attending to all those myriad detail that make life easier for us graduate students.

To all my friends in the Space Systems Lab, who have helped along the way I thank you. Dr. Graeme Shaw, the old, wise graduate student, your advice was always dead-on, and your example was inspiring. Dr. Edmund Kong, you've always been helpful, always a great office-mate. And thanks for the tea. I cannot forget to mention Dr. Cyrus Jilla. You have always been a bastion of enthusiasm and an example of diligence. Although we have not exhausted all of the culinary delights of Boston, we've made a good start. Mr. Michel Ingham, a fellow Canuck (and Leslie), you are truly a good friend (and the lab's social convenor).

Becoming a senior grad-student came as a surprise but the newer generation is as important as the old. Ms. Alice Liu, thank you for all the invaluable consultations and the constant warmth. Good luck and best wishes for next year. Ms. Rebecca Masterson, I am sorry that the Red Sox will never again win a world series, but I still thank you for all the LaTeX help. Mr. Andrew Radcliffe, my GFLOPS compatriot, thanks for your contributions. You're a great office-mate. Mr. Alvar Saenz Otero, you are truly the spirit of MIT embodied (and a great inline-skater).

Mr. Marl Hilstad, it's good to find another as crazy about doing hiking as I am. We've still got 24 of the White Mountains left. There's no time to waste! By the way, when are we doing the real mountains?

Ms. Karen Marais: coffee, philosophy, or your attentive editing; I value them all. You were a great office-mate and are a good friend. Through cynicism and enthusiasm, I think we see MIT in the same way. Any more puppet plays lately?

To all the other friends here, this has been a fantastic time.

I can't help but thank my parents, Dr. & Mrs. Wayne and Rosemary Enright. You've both been extremely instrumental in getting me where I am today. You've given me the drive to reach for my goals. Most of all you gave me an environment of love and support. I would

like to extend a special thanks to my dad who serves as a fine expert on all things computer science. You always know which way to point me.

Here we are; I've saved the best for last. The biggest, most basic thanks of all must go to my wife and best friend, Sarah Enright. You always give me an ear to listen, a chance to remain sane, and someone to lean on. I can't list all the things that you've helped me with, especially in the last few months. I know what you've done and I appreciate every second of it. Thank you.

NOTE: The author would like to thank the sponsor of this work: The Air Force Office of Scientific Research for Generalized Flight Operations Processing Simulator - Contract F49620-99-1-0217, under the technical supervision of Dr. Robert Herklotz.

Biographical Note

John Enright was born to Wayne Enright and Rosemary Enright (née Saville) on June 13, 1974 in Toronto, Ontario, Canada. Growing up in the suburb of Scarborough, John entered the gifted program of the Scarborough Board of Education. Graduating from Woburn Collegiate (high school) in June of 1993, John entered the Engineering Science program at the University of Toronto. John was the recipient of a Canada Scholarship and a UofT Scholarship. Specializing in aerospace engineering John worked several summers as a summer student at the University of Toronto Institute for Aerospace Studies (UTIAS) in the Space Robotics Laboratory. Graduating with Honours in 1997, John moved to Boston to enrol at MIT for graduate studies. He completed is Masters in Aeronautics and Astronautics in 1999.

Selected Publications:

Enright, J., Sedwick, R., Miller, D., "High Fidelity Simulation for Spacecraft Autonomy Development", *Canadian Aeronautics and Space Journal*, Dec. 2001 (Forthcoming).

Enright, J., Jilla, C., Miller, D., "Modularity and Spacecraft Cost", *Journal of Reducing Space Mission Cost*, Vol. 1 Num. 2, 1998.

Enright, J., Sedwick, R., Miller, D., "An Enabling Development Environment for Spacecraft Autonomy", *AAAI Spring Symposium: Robust Autonomy*, March 26-28, 2001, Stanford, California.

Enright, J., Sedwick, R., Miller, D., "An Evolutionary Approach to Engineering Spacecraft Flight Software", *IASTED Applied Informatics 2001: Software Symposium*, Feb. 19-22, 2001, Innsbruck, Austria.

Enright, J., Sedwick, R., Miller, D., "High Fidelity Simulation for Spacecraft Autonomy Development", *ASTRO-2000 11th CASI Conference on Astronautics,* Nov. 6-9, 2000, Ottawa, Ontario.

Enright, J., Sedwick, R., Miller, D., "Information Architecture Analysis and Optimization for Space-Based Distributed Radar", *AIAA Space Technology Conference*, Sept. 28-30, 1999, Albuquerque, New Mexico, (AIAA 99-4551).

Enright, J., Carrol, K., "Laser Power Beaming for Lunar Pole Exploration", *ASTRO'98 10th CASI Conference on Astronautics*, Oct. 26-28, 1998 Ottawa, Ontario.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Space engineering is simultaneously one of the most innovative and conservative technical fields. Designers are asked to produce revolutionary systems, yet at the same time they must be cautious, since mistakes in design and manufacturing are extremely costly. With satellites routinely costing on the order of at least $100 million (and some even several billion dollars), there is great reluctance to stray very far from proven solutions. This trend is particularly apparent in the development of spacecraft *flight software (FSW).*

Innovative mission concepts stretch the capabilities of conventional flight software to their limits. Greater software complexity promises both greater performance and greater risk. In this thesis we examine some of the issues involved in the confluence of advanced flight software and distributed satellite systems. Specifically, we present an approach to managing software communications[1] complexity, suitable for the high-reliability, real-time environment of spacecraft systems.

1.1 Motivation

In the design of complex systems, software requirements are often ambitious and fluid. These trends are difficult to avoid, since the modern computer is the ultimate *general pur-*

1. The term 'communications' within this thesis refers to the exchange of data between software components and not just a satellite 'communications subsystem' (e.g. receiver, transmitter, antenna, etc.).

pose machine. There are few intuitive limits to the tasks assigned to software. Functions difficult to implement mechanically could be embedded into an electronic control system, or stubborn electrical noise might be digitally filtered, rather than eliminated at the source. It is also not uncommon to see software requirements changing long after hardware specifications have been fixed [Shore, 1986]. Functional flexibility has its drawbacks, since additional responsibilities create sub-system couplings that, in turn, increase overall system complexity. The greater the system complexity, the easier it is to introduce errors in design [Leveson, 2001]. Undiscovered system-level software errors can lead to catastrophic failures, such as the first launch of the Ariane 5 [Lions, 1996].

Building software that works most of the time is relatively simple; building software that works all the time is very difficult. Advanced software capabilities require innovative development methodologies to ensure success.

1.1.1 Promise of Software

Flight software has the capability of performing many tasks, both wondrous and mundane. Advanced mission concepts are currently being developed that would autonomously explore the seas thought to lie beneath kilometers of ice on Europa [Doyle, 1998]. Even smaller missions, such as the Pluto Fly-by concept, contemplate giving the on-board software the capability of prioritizing and planning its own scientific observations. The challenges of these systems are truly daunting.

Researchers have recognized for several decades that space-systems can benefit considerably from 'smarter', more capable software [Marshall, 1981]. Concerns about reliability have limited the applications of *autonomy* to situations in which traditional methods have proved completely inadequate. Suitable applications would include the critical phases of an interplanetary mission where propagation delays prevented traditional ground-based commanding. When employed, autonomy techniques were applied in a minimalist fashion; engineers used only as much sophistication as was necessary. Recently, interest is

growing in expanding autonomy's role to routinely control a wider range of spacecraft activities.

Not all goals of flight software development are so lofty. Future plans for the Global Positioning System (GPS) navigation satellites call for advances in on-board software. Daily operation of the satellite constellation is presently a very labour-intensive task. Next generation GPS satellites must maintain their effectiveness for several weeks or months without input from the ground [Fisher & Ghassemi, 1999]. Instead of human operators they must rely on their own software and on communication with their peers. Even more prosaically, the history of the Galileo spacecraft provides a clear example of the flexibility of software. Crippled by the failed deployment of its High Gain Antenna, this multi-billion dollar mission still returned amazing science data, due largely to extensive modifications to the onboard software's data processing and communications routines [Marr, 1994].

Flight software fills many roles in space systems. The above examples demonstrate some of the potential flexibility. Software in the Europa mission adds wholly new capabilities. The enhancements to the Pluto mission facilitates operations made difficult by long time delays. It can be used for automation and even, to a degree, for upgrading and repair. Unfortunately, these capabilities do not come without a price. Although advanced software promises a large degree of flexibility, it also imposes an ample amount of risk. Project managers shoulder the responsibility for mission success and must be particularly careful about adopting new, unproven technology.

1.1.2 Distributed Satellite Systems

One focus of research at the Space Systems Laboratory at the Massachusetts Institute of Technology, is the application of distributed approaches to space systems engineering. Starting with the basic principles of distributed systems, we have examined some of the resulting implications for space mission design. The chief considerations are:

- Architecture. For millennia, it has been understood that "*form follows function*" [Vitruvius, 1960]. Missions using sparse aperture sensors [Das &

Cobb, 1998] [Beichman, et al, 1999], or providing global communications [Garrison, et al, 1995] are immanently suited for distributed implementations.

- Composability. Since distributed satellite systems are composed of separate spacecraft, they can be tested, deployed [Miller, et al, 2001] or replaced [Shaw, et al, 1999] progressively.
- Fault Tolerance. Many traditional systems rely on a single central component. Mission assurance depends on making this device (or satellite) very reliable. Distributed systems, in contrast, are designed to exhibit graceful and gradual degradation in performance as component satellites fail [Shaw, 1998].
- Extensibility. It is usually easier to augment the capabilities of distributed rather than centralized missions [CDIO, 2001]. This principle applies during the operation of a system as well as during design when requirements may change.

When we consider the software requirements for distributed spacecraft systems, parallels can be drawn to the history of terrestrial computing. As popular design practice moved from centralized to distributed architectures, the importance of communications grew immensely [Verissimo & Rodrigues, 2001]. Communications pathways were needed, not only to relay information, but to manage system coordination as well.

1.1.3 Communications Complexity in Software Systems

One of the principal characteristics of systems engineering is the management of complexity [Booton & Ramo, 1984]. The entire discipline grew out of the difficulties encountered when building large, elaborate systems; projects that resisted simple comprehensive designs. Complexity management is doubly important in software systems, since software is frequently used to deliberately create couplings between independent components [Shore, 1986]. For example, consider the role of anti-lock brake systems in modern cars. In a traditional brake system, pressure on the brake pedal is translated, fairly directly, into friction force on the wheels. With anti-lock brakes, there is now a software-based feedback loop that modulates the brake actuation based on the observed wheel speed. Coupling these two behaviours together provides a benefit to the driver, but makes the whole system more complex.

Modern software engineering emphasizes the importance of modular design. Common functional elements are grouped together, and clearly delineated interfaces describe allowed interactions. Adding new functions to a system typically adds more modules. With new modules, come new interfaces, whose numbers can grow non-linearly. This compounds system complexity. The situation becomes even worse when the system is distributed. Within a CPU, communication between processes is an abstraction; in a distributed system, physical communication is a necessity.

Despite the popularity of various software engineering methodologies, building complex, reliable software systems is still very difficult. That software was directly responsible for the failure of several recent space missions suggests shortcomings of the prevailing development culture [Leveson, 2001]. Interconnections between software components are one of the leading causes of errors in complex software. Studies documenting the validation of spacecraft show that interface errors account for 20-35% of all flight software flaws [Lutz, 1992]. Clearly, developers must devote much of their attention to the correctness of software connectivity.

In his book, *"Augustine's Laws"*, former Lockheed-Martin CEO, Norman Augustine proposes the following maxim:

> Law XLVIII: The more time you spend talking about what you have been doing, the less time you have to spend doing what you have been talking about. Eventually, you spend more and more time talking about less and less until finally you spend all your time talking about nothing. [Augustine, 1997]

With a little creative reinterpretation, what was true for management, can be applied to software. Each additional software connection that must be managed by the developers creates overhead that detracts from productivity. The more time you spend managing interfaces, the less time you spend implementing function.

Communications complexity can affect flight software on several levels. Concurrent tasks on a single processor must share the available computational resources while communi-

cating effectively with each other. Within a satellite, managing cooperating processors can also be challenging [Crew, 2002]. Coordinating separated spacecraft can be even more problematic, especially if real-time performance is required. When the Iridium communications system came online, users' calls were often dropped, as software attempted to pass control of the call to another satellite [Mills, 1998]. As distribution, either within, or between satellites, becomes commonplace, we expect to see a demand for complexity management in software communications. We suggest that the aerospace industry should follow the example of terrestrial distributed systems and use abstract communications services to reduce the burden of communications complexity in spacecraft software.

1.2 Middleware for Complexity Management

When functional modules are connected together, assumptions must be made about the way in which interactions take place and how information is exchanged. Depending on the computing architecture, the user may employ shared variables, inter-process messages, function invocation, or a combination of the above. When software modules are distributed across separate processors, the possible interaction mechanisms multiply. If the details of each software interconnection must be handled individually, the design and implementation of the software is made much more complex. Furthermore, the migration of certain components to different processors, or the addition of extra functions, may require extensive modifications to the existing software. This is the problem that middleware addresses.

Middleware commonly refers to software that manages interconnections between the user's software applications (Figure). A standardized interaction mechanism abstracts away some details of the connection. Middleware operates in networked environments and provides interface *transparency* to the user. Thus, the designer needs to specify that Module-A is connected to Module-B, but they need not worry about the precise implementation, or even which processors the modules run on. Middleware may be integrated into the operating system, or it may exist as a separate "service" layer.

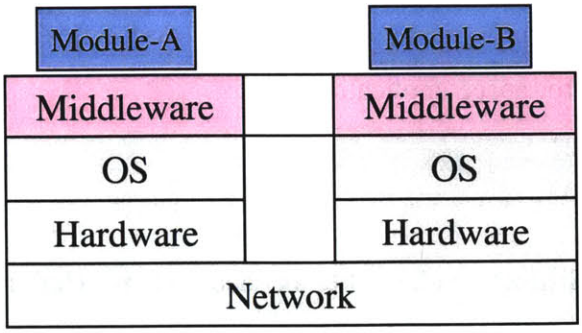


Figure 1.1 The Middleware concept. Middleware software allows user applications (modules A and B) to transparently span processor boundaries.

Designers of middleware systems usually select a particular interaction model as the basis for their system. Some examples include: distributed objects (e.g. CORBA, DCOM) [OMG, 2000] [Hortsmann & Kirtland, 1997], message passing (e.g. MPI) [Snir, et al, 1995], transactions (e.g. Tuxedo) [Hall, 1996], and distributed shared memory (e.g. BRAZOS) [Speight & Bennett, 1998].

When choosing a middleware system, users must be mindful of the desired level of abstraction and the underlying conceptual model. Generally, abstract systems such as the popular *Common Object Request Broker Architecture* (CORBA) hide many implementation details from the user, and greatly reduce interface management tasks. The code and data that implements a CORBA object may reside on a remote machine, but users can create, manipulate, and invoke the objects as if they were local. Conversely, concrete mechanisms such as message passing provide the user with a precise mental model of what the system is doing, but leave more tasks to the user. Any particular interaction metaphor will facilitate certain applications, but may hinder others [Lee, 2000]. Object-based methods, despite their popularity, are not always a good design metaphor for embedded systems [Wright & Williams, 1993].

Middleware choices affect testing and validation as much as they affect design. Each middleware implementation provides the developer with different performance guarantees. Deterministic memory and processing overhead, bounded temporal performance, and

compatibility with existing software engineering techniques are all vital to real-time applications. Thus, systems with the best combination of service flexibility and performance guarantees will be ideal for space applications.

1.3 Objectives

Spacecraft[1] represent a distinctive class of embedded systems. High reliability is essential, and fail-safe operation may be difficult to ensure. Consequently, engineers require a fine degree of control over how the software behaves. Physical interaction with the hardware is essentially impossible, thus faults must be handled automatically or remotely. Lastly, complex spacecraft have elaborate control systems with hard real-time requirements. This makes deterministic and efficient temporal performance a priority. As the demands of distributed implementations and advanced capabilities aggravate system complexity, new tools and approaches will be required to manage software interactions. The primary objectives of this thesis are to develop flight software engineering approaches that will:

- Reduce unproductive workload of managing software interfaces.
- Reduce software complexity to enhance reliability or mitigate risks of advanced capabilities.
- Respect the constraints of the flight software engineering domain.

We approach these goals by developing middleware for space applications. State-of-the-art middleware systems (Section 2.3) typically provide flexible services, and only modest real-time assurances, or extensive real-time validation, but limited complexity management. Advanced space systems need both qualities. Our middleware provides *publish-subscribe* services with hard real-time bounds on latency and jitter.

Establishing a tool's suitability for the space domain is not a trivial task. First, the middleware concept must be developed and implemented. Second, extensive validation and characterization must be used to ensure the safety and predictability of the system as well as to

1. Our study focuses on spacecraft software but this approach would likely benefit aircraft software as well.

assess the 'cost' (e.g. processing, memory, performance) of the system. Finally, we must demonstrate, through examples, how the tool can be used to manage software complexity.

1.3.1 GFLOPS

Flight programs are difficult to come by, and not surprisingly, difficult to freely experiment with. Even if we were assigned the task of developing middleware for a particular mission, it is unlikely that we would be afforded the freedom required to explore the range of software concepts valuable to academic research. Instead, we adopt a more modest but versatile application: spacecraft simulation.

This research constitutes part of the *Generalized FLight Operations Processing Simulator (GFLOPS)* program. GFLOPS's goal is to produce a software testbed suitable for high-fidelity, real-time simulation of distributed spacecraft systems. GFLOPS allows the user to develop sophisticated simulations and prototype flight software in a hard real-time environment. During development of the testbed, we identified the need for a software environment that would support rapid application development. High-performance embedded middleware was our solution.

1.3.2 GRRDE

Our middleware system is called the *GFLOPS Rapid Real-time Development Environment (GRRDE)*. The conceptual model behind GRRDE is a *publish-subscribe* architecture. Software module interfaces are specified in terms of particular classes of *data-products*. An output data-product is said to be *published*. Modules needing these data as inputs, can request a subscription from the GRRDE middleware. The data in question are then automatically delivered to the subscriber. Subscribers can request updates at preset intervals or whenever the source value changes. These services provide sufficient abstraction to reduce the effort needed to manage module connections, yet are concrete enough to maintain consistent mental models of operation. The publish-subscribe model also provides a

more natural mapping than objects for many common flight software functions. Examples include digital control systems, monitoring, sequencing, and data processing.

Although a spacecraft simulation is not the same thing as an actual spacecraft, we have endeavoured to build into GRRDE consideration of its final destination. Every attempt was made during development to integrate the requirements of embedded systems. Nevertheless, GRRDE is still a 'pre-release' system. All essential technologies have been validated, but polishing the system to the point of commercial viability would require additional resources.

1.3.3 Validation

Software for use in embedded real-time environments must be tested more extensively than everyday applications for personal computers. Since so much depends on "getting it right the first time," flight software engineering is an extremely conservative field. To put this in perspective, consider that many spacecraft still use cyclic executives to control processing rather than process-based operating systems. It is incumbent on us, as designers of a new tool for space systems, to provide strong assurances about the performance of our product.

We approached the validation of GRRDE from three perspectives: conventional software testing, formal analysis, and traditional real-time analysis. Software testing consisted of thorough static analysis, example cases, and stress tests. This testing was used to identify and correct bugs, as well as measure temporal performance parameters such as timing jitter. We supplemented the online testing with formal analysis. Using the *General Timed Automata (GTA)* [Lynch, 1996] modelling technique, we were able to prove safety and performance properties of our algorithms. Included in these results are upper bounds on system jitter. Finally, we suggest strategies for applying real-time analysis techniques, such as *rate monotonic analysis*, to GRRDE-based systems. This combined approach allows the designer to build confidence in the GRRDE services and predict system performance.

1.3.4 Applications

The primary goal of GRRDE is to provide a complexity management tool. Validation establishes its suitability for embedded applications. To complement this effort, a number of application studies were conducted to assess its usability. As a baseline mission, we developed a detailed simulation of the TechSat 21 experimental radar satellites using GRRDE. Smaller studies examined the use of GRRDE in conjunction with other advanced flight software techniques.

TechSat 21 is a distributed satellite demonstrator mission. This experimental radar program uses a cluster of small satellites and interferometry techniques to do highly accurate ground and air moving target indication (GMTI/AMTI) [Das & Cobb, 1998]. The spacecraft must be capable of complex on-board processing and high accuracy formation-flying. TechSat 21 represents a challenging and complicated mission that will highly stress on-board processing capabilities. Our simulation captured orbit and attitude dynamics and control, as well as the complex radar processing. We found that GRRDE was a significant asset in the design and implementation of this large system.

The complexity management properties of GRRDE can be viewed in two ways. First, that lower software complexity makes a given system safer and more reliable. Alternately, this complexity management can be employed as an enabler; i.e to permit software that normally would be too complex or risky.

We chose two examples to examine this latter property. The first used automatic code generation tools, to directly create GRRDE-enabled control systems from Simulink models. The information mobility provided by the middleware complemented the rapid-development capabilities of our code generation tools. Our final study integrates GRRDE with an autonomous fault diagnosis engine. The use of GRRDE middleware allows the supervisory functions of the diagnostic engine to be minimally intrusive with respect to existing software. The results of these studies are encouraging and show the effectiveness of GRRDE during development and integrations of complex software systems.

1.4 Outline

We begin with a discussion in Chapter 2 of previous work and general theory of real-time systems, flight software, and middleware. Chapter 3 presents the engineering of the GFLOPS testbed. The structure of the GRRDE middleware is developed in Chapter 4. Formal analysis in Chapter 5 proves essential system properties and the on-line testing in Chapter 6 backs up the offline validation with direct measurements of system behaviour. The system-level ramifications of GRRDE are detailed in Chapter 7. System design and characterization is supplemented by several application studies, notably an exercise in automatic code generation (Chapter 8), fault diagnosis (Chapter 9), and large scale simulation development (Chapter 10). We conclude with a summary of our accomplishment and suggestions for future work.

Chapter 2

BACKGROUND

The primary objective of this thesis is to provide a tool that will help manage software communications complexity in advanced space systems. In the previous chapter, we discussed current directions in space flight software and the promise of distributed satellite systems. The GFLOPS Rapid Real-time Development Environment (GRRDE) provides a number of helpful, abstract services, reinforced with extensive analysis. This chapter places our engineering efforts in perspective with background information about space systems, embedded software and middleware. We present several examples of convergence in space and terrestrial embedded systems that support our expectation of the emerging need for these tools.

2.1 Flight Software Engineering

Writing flight software is one of the most difficult varieties of software engineering. Flight software, like other spacecraft components, must meet high performance standards and be extremely reliable. Even conventional embedded systems do not possess the particular combination of design constraints that makes space software so challenging. Partly due to culture, partly due to necessity, spacecraft software, though punctuated with innovations, is nonetheless fairly staid and conservative. The state of the art in embedded and distributed systems has advanced with modern techniques. Flight software engineering might follow their example and capitalize on these advances too.

2.1.1 Space Engineering

Space system applications add an additional layer of considerations to the process of software engineering. The distinctive characteristics of the space industry make software development difficult. These challenges not only apply to on-board flight software, but to ground support software as well. High risks and high costs create a very cautious design mentality, punishing environments strain processor capabilities, and low product volumes necessitate designs both correct and fault tolerant.

Few other engineering environments are as demanding as space systems. Once launched a spacecraft is on its own for as much as a decade or more. Except in very specialized circumstances, such as the Hubble telescope, repairs are out of the question. For a commercial satellite, a failure may mean bankruptcy. Consequently, the development environment is very risk averse. The benefits of any new technology must be carefully weighed against the potential for introducing failures. Unless critical for mission success, new techniques are unlikely to be adopted.

Harsh radiation makes the orbit environment an unfriendly place for modern electronics. Any device intended for launch must either be protected with heavy shielding, or subject to extensive and costly 'radiation hardening'. Consequently, the state of the art in spacecraft processors is often a decade or more behind their terrestrial counterparts.The dangers of radiation are not limited to the direct degradation of components. High energy particles are known to flip bits in memory or microprocessor registers. Some actions can be taken to detect and correct most of these errors, but undetected problems of this type can be very serious. The Total Ozone Mapping Spectrometer-Earth Probe (TOMS-EP) spacecraft was nearly lost when a high-energy particle scrambled computer calculations [Robertson, et al, 2000].

Other products, such as aircraft or power plants, share the same or greater criticality as spacecraft, but certain factors set the space industry apart. All large systems must undergo a period of qualification and testing before deployment, but it is very difficult to test space

hardware in the same environment in which it will be deployed. Hence, greater reliance is placed on component and subsystem testing. Careful systems analysis must be employed to foresee undesirable interactions between system segments. Furthermore, the relatively low volumes of spacecraft produced make it difficult to work out all the problems with a design before it is obsolete.

An understanding of the constraints on embedded systems and the particular characteristics of spacecraft design establish the necessary background for enhancing the process of FSW development. The next few sections build upon this foundation and introduce the particular formalisms and tools introduced in the GRRDE. The aim of these advances is to reduce the development time and increase the quality of FSW.

2.1.2 Real-Time, Embedded Software Engineering

Common misconceptions about real-time computing abound. Many people, even in technical fields, equate real-time with 'fast' [Zita Haigh, et al, 2000]. Although efficiency of implementation is certainly important [Rajkumar, et al, 1995], fast code alone does not define real-time software. Above all else, real-time systems must be predictable. The requirement spans temporal performance, memory and system-level interactions.

Time of course places a vital role in the operation of real-rime systems. Effective software must be both predictable and efficient. In most cases real-time software can be classified [Kopetz, 1997] according to how stringent the temporal determinism must be. *Hard real-time* systems are most stringent about the system's predictability under all foreseeable operating conditions. Bounding worst-case behaviour is critical and low variability (jitter) is important. Violating the temporal bounds can have extreme consequences. At the other extreme is the notion of *soft real-time* systems. These systems value predictability, but can usually tolerate a fair degree of temporal variance in performance, especially under high-load conditions. Some other variants are *firm real-time* systems (strict deadlines, less criticality) and *anytime* systems (minimal response temporally guaranteed, high performance results computed if time permits). In hard-realtime systems, algorithmic efficiency is only

a consideration after bounded execution time has been established. Code optimization can be used to save money (e.g. with a cheaper processor) [Stewart, 1999] or meet CPU utilization budgets, but should not be regarded as an end in itself.

Memory is another carefully rationed resource in embedded systems. A necessary part of guaranteeing predictable performance is ensuring that memory usage remains bounded [Stewart, 1999a]. Many embedded systems have tightly constrained memory budgets. Space systems, even in the last decade, frequently have less than a megabyte of on-board memory [Wagner, 1998]. In most cases dynamic memory allocation is forbidden by design standards, since running out of memory would be disastrous.

Finally, we come to the most important, defining feature of real-time systems: testing. All the characteristics that typify embedded and real-time systems, especially for space applications relate back to the need for testing and analysis. Since failure in safety-critical software can lead to loss of both property and lives, designers have professional, ethical, and legal responsibilities to make sure that products are tested adequately. Temporal performance must be tested. Memory requirements must be tested. Assembled systems must be tested, not just against the requirements, but against the intentions behind them [Leveson, 2001]. Although there is never time to test everything, a well conceived test program is critical to mission success.

In the interests of clarity, let us stop and consider a manner of vocabulary. The terms 'real-time' and 'embedded' are used interchangeably in this thesis. Although the meaning of the terms are similar they are not always the same thing. Kopetz [Kopetz, 1997] identifies the following classes of real-time systems:

- Embedded systems. This class of software systems involve the operation of a particular, special purpose device or 'intelligent product'. Applications range from electric shavers, to microwaves to medical diagnostic equipment. Configuration and function are fairly static.
- Plant automation. These systems are typically larger and often distributed. As the name suggests, they typically govern the operation of manufacturing

machinery. Digital feedback control is common in these systems. Produced in low volumes, these systems are heavily customized for each installation.

- Multimedia Systems. In recent years, as internet popularity has increased, interest in high-quality soft real-time systems for video and audio delivery. This class of systems is not relevant to us.

Space systems[1], the focus of this thesis, combine characteristics of both plant automation and embedded devices. Satellites are device-centric, static in (hardware) configuration and involve a large amount of process control. Thus, the labels of 'embedded' and 'real-time' are equally valid for spacecraft.

2.1.3 Traditional Approaches to Flight Software

Flight software, like other embedded programs are designed for a particular processing model. Such a model dictates the environment in which the source code will run. We will call this environment the operating system (OS), even though properties offered by the environment may be much less comprehensive than those of most personal computers (PCs). The primary task of this operating system is to allocate processor resources to the various computational functions that must be performed.

Execution Models

During design, software responsibilities are divided into manageable pieces. Most early software responsibilities consisted of doing the same job repetitively, in a regular cycle [Bernstein, 1996]. Each *periodic* task can be then be programmed as its own subroutine. The earliest types of operating systems know as *cyclic executives* would sequentially execute each task[2] in a perpetual, timed loop (Figure 2.1).The logic might match this simple pseudocode:

```
loop
  do_task_A()
  do_task_B()
  do_task_C()
```

1. This observation could be applied to aeronautical systems as well.
2. Although some make a distinction, we shall discuss tasks, processes and threads as equivalent concepts.

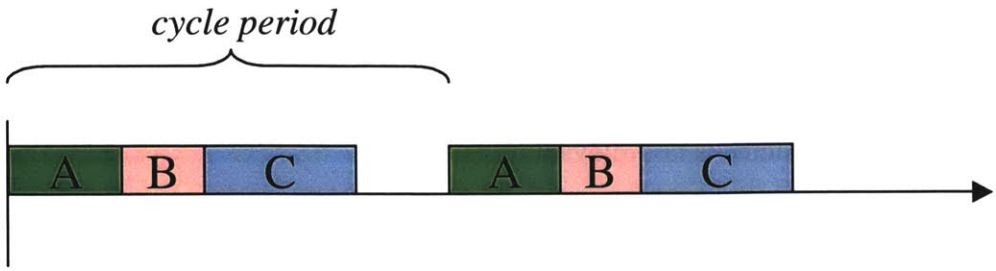


Figure 2.1 Execution of a cyclic executive.

```
  wait_until(next_cycle)
end loop
```

Unfortunately, a number of problems plague these systems [Locke, 1992]. First, all task periods must be a multiple of the shortest task. The greater the variation in cycle times the more complex the loop structure must be. Second, this arrangement makes it difficult to separate the relative importance of different tasks. If one task takes too long, other, more important processes, may not finish their work in time. Third, tasks with long computation times must be divided into sub-tasks of more manageable duration. Generally, this approach, while simple, scales poorly as software functions expand.

Concurrency is a concept that removes many of the restrictions of cyclic executives, but imposes difficulties of its own. In concurrent or multi-processing systems, each task is viewed as an entity executing in parallel with the other tasks in the system. This is merely a conceptual model, since single processor systems cannot maintain multiple threads of execution. The illusion of concurrency is provided by a run-time software component that selectively gives each task control of the processor (Figure 2.2). Although this added flexibility avoids the complexity of elaborate cyclic systems, we lose the ability to tell exactly what all other tasks are doing at a given instant. Thus, if more than one task shared a certain resource (e.g. some memory or an I/O device), step must be taken to avoid two tasks trying to use it at once.

Coordinating processor access for the various tasks is not always straightforward. On one hand, we wish this scheduler to add minimal overhead to the system, yet we want it to be

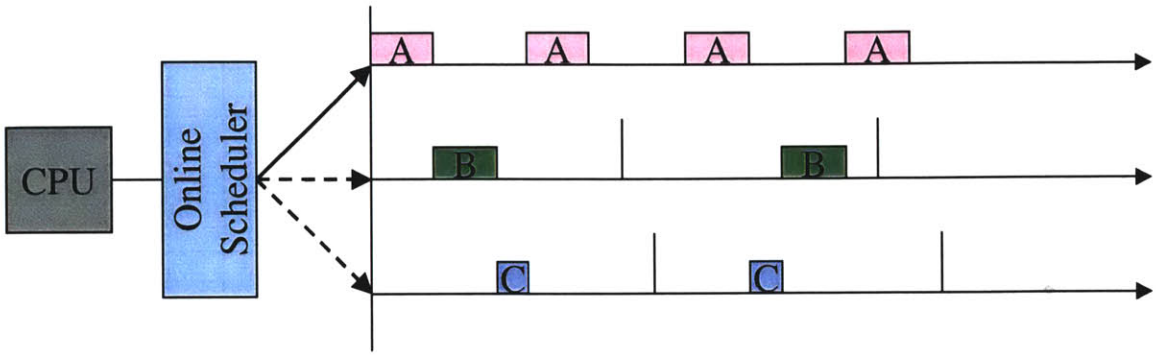


Figure 2.2 Conceptual model of concurrent processing.

as effective and efficient as possible in switching between tasks. Table 2.1 details some of the arbitration mechanisms and their relative heritage in space systems. Time-sliced systems are probably the most common mechanism employed in spacecraft after cyclic executives [Malcom & Utterback, 1999]. These systems eliminate many of the shortcomings of cyclic executives, and are still inherently simple to design and analyze. Preemptive, dynamic (priority-based) scheduling is probably one of the most popular schemes for terrestrial systems. One of the primary benefits of these systems is the ability to increase system responsiveness to aperiodic tasks, i.e. those triggered by events rather than the passage of time.

Of the static priority schemes, *rate monotonic priority assignment* (RMPA) [Liu & Layland, 1973], is probably the most popular. The RMPA scheduler is simple to construct and has low computational complexity. It can guarantee deadlines for any set of tasks with a net processor load of less than about 70%, and most[1] task sets with net utilization of 88% or less [Lehoczky, et al, 1989]. This technique has been studied extensively in the literature for both periodic and aperiodic tasks.

Although this technique is quite popular in terrestrial applications, it has still not been universally adopted in space systems. Conversations with flight software engineers suggest

1. I.e. for most random sets of tasks with utilization of 88%, RMPA guarantees scheduability. For any *particular* set of tasks, we can definitively assess RMPA scheduability.

TABLE 2.1 Task Scheduling Methods

Task Switching	Scheduling	Operation description	Space Heritage
Non-Preemptive	Static	This is essentially a cyclic executive	Extensive (e.g Space Shuttle [Spector & Gifford, 1984])
Preemptive	Static	Processes are assigned fixed percentages of execution time. When a time-slice expires, execution of the task is suspended until the next time-slice.	Moderate (e.g. Hete-2 [Crew, 2002])
Non-Preemptive	Dynamic	Tasks may execute to completion but selection of next task to run is performed at run-time.	None known
Preemptive	Dynamic	Task selection usually based on dynamic or static priority system. Highest priority enabled task gets the use of the CPU. Can be very efficient.	Some (e.g. Mars Pathfinder [Stolper, 1999])

that this due mainly to risk tolerance. Using commercial operating systems in spacecraft is now quite commonplace [Malcom & Utterback, 1999] and interest in preemptive multitasking is growing [Kolcio, et al, 1999].

Terrestrial embedded systems have generally adopted these flexible process models. Moreover, researchers are increasingly concerned with execution, not just on uniprocessor systems but in real-time distributed networks [Sha, et al, 1994]. It is generally acknowledged that robust, flexible communications services are essential areas of development to manage the complexity of these systems [Kopetz, 2000] [Lee, 2000]. Middleware software has been used successfully to add communications abstraction to conventional distributed systems. There is increasing interest in applying it to aeronautical [Harrison, et al, 1997] and missile systems [Clark, 1991]. We conclude from this trend that space applications will likely follow.

2.2 Distributed Systems and Middleware

Many factors have contributed to the rise of distributed computing. Traditionally, most distributed systems used for banking, aircraft reservations, corporate databases, etc., were distributed for fault tolerance and traffic management. This represents the classic principles of distribution. Lately, however, distributed ubiquitous computing has taken on a life of its own. Fast, capable personal computers have de-emphasized the need for powerful local servers [Bernstein, et al, 1987]. Connectivity is everywhere — Local Area Networks, Wide Area Networks, Wireless Networks, etc. — and is growing more complex and unpredictable by the day. Communications between all of these heterogeneous components requires some help.

In Chapter 1 we discussed the role of abstraction in communications services. In this section we explore how abstraction facilitates *transparency*. Transparency is an essential feature of distributed systems and can be used in a number of ways. Choosing a middleware system means choosing a level of abstraction and a communication metaphor. Several common approaches are examined in the context of embedded and space applications.

2.2.1 Middleware and Transparency

The goal of middleware systems is to provide communications abstraction for the user or application developer. Abstraction relieves the user from the responsibility of managing the details of implementation and allows one to concentrate on what is important. Disk drive structures and directories are abstractions that many people use every day. Communication abstractions remove the particulars about connections between software components. Thus, the developer may spend his time implementing functional, rather than physical, coordination.

Take for example the simple system shown in Figure 2.3. As a developer you are charged with creating the software component A which must interact with a preexisting component B. Transparency is a type of abstraction that allows the developer a consistent inter-

face to another software component, regardless of how or where the other component is implemented. In our example, the interface between A and B is transparent if our interactions remain for many possible implementations of B. Many types of transparency are possible, and some are more important to distributed systems than others.

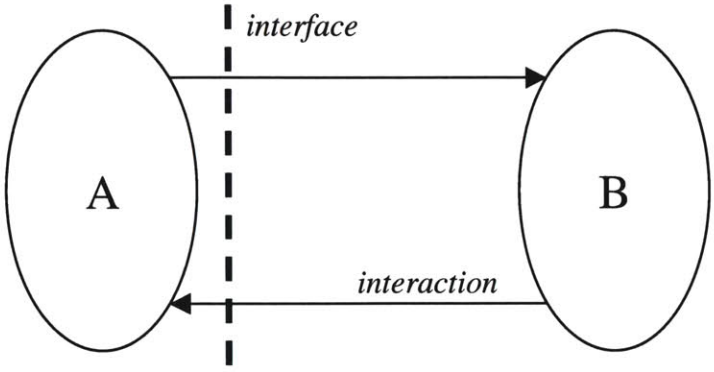


Figure 2.3 A very simple illustration if software interaction.

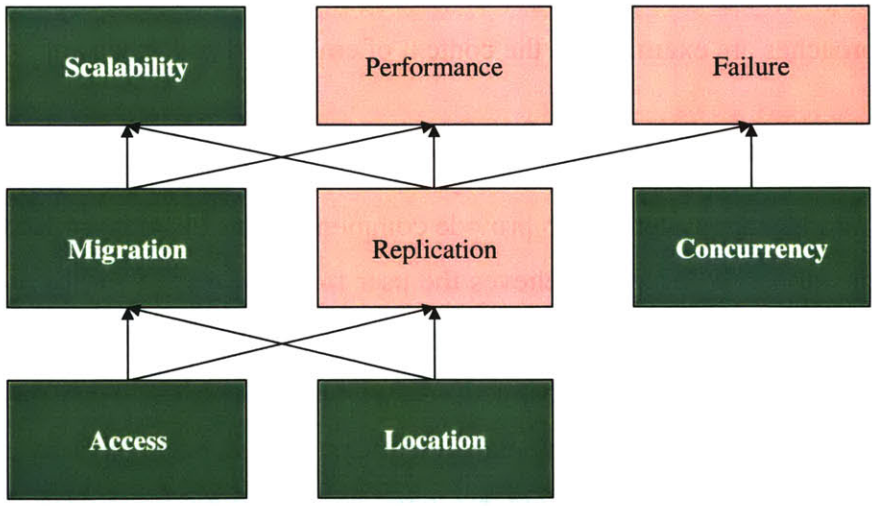


Figure 2.4 Dimensions of Distributed Systems Transparency. (Taken from [ISO/IEC, 1996] Dimensions highlighted in green (dark-boxes) are priorities identified for embedded space systems.

An International Standards Organization study on Open Distributed Processing [ISO/IEC, 1996] identified eight *dimensions of transparency* (Figure 2.4). These areas are interrelated and build upon each other. They describe properties of distributed systems of varying

complexity and importance. We can briefly state the characteristics of each type of transparency [Emmerich, 2000a]:

- Access. Software can interact in the same manner (i.e. exchange information), regardless of where it is (local or remote) or how it is implemented.
- Location. Software components can find other components regardless of location.
- Migration. We can move a component from one host to another without affecting interaction.
- Replication. Multiple copies of the destination component may exist on different networked machines.
- Concurrency. Users need not be aware of ongoing interactions between other software and the destination component.
- Scalability. System can grow substantially (more connections, more components), yet maintain the same basic interaction mechanism and architecture.
- Performance. Mechanism by which performance is obtained is hidden from users. May include dynamic load balancing
- Failure. Destination components may fail without affecting consistency of local component.

Not all of these properties can (or should) be found in all distributed systems. The suitability of different varieties of transparency depends on the application. For embedded space systems, this is especially true. In some situations, a particular type of transparency is not a priority; in other situations there is considerable justification for not making particular interactions transparent. After a point, too much transparency works against most real-time and embedded systems, since fundamentally, the purpose of these systems is centered around the operation of a particular device. When you load a web page, you don't care which one of the *Wall Street Journal* web-servers sends you the information, but when you print a document, you want it to appear on the printer next to your desk.

We make the following assumptions about transparency in space systems. Access and location transparency are generally useful in embedded systems. Migration transparency is useful within the system's requirements for latency and timing jitter. For space applications, migration might allow certain software functions to be transferred from ground to

the satellite [Jones, et al, 1998]. Replication is probably not a priority due to the fairly localized nature of most satellite systems. Some missions may require this more than others (e.g in telephony systems, the handset *should* interface transparently to any satellite overhead). Concurrency is fairly important when several layers of control systems must be monitored and coordinated. At the highest level, only scalability is applicable to most space missions. Even still, the degree to which we need scalability is limited. Making allowance for anticipated services, or system upgrades is sensible, but over-design is wasteful. Another important consideration is range of scalability needed. Terrestrial systems may scale by several orders of magnitude, but the requirements for space systems is typically much less. We do not consider performance transparency of primary importance to space systems, but it may be useful to distributed telecommunications systems. Finally, we specifically omit failure transparency due to the nature of real-time systems. This is not to say that space systems should not be fault-tolerant, rather that they should rarely be transparently so. Designers of hard real-time systems need explicit control of the recovery or compensation mechanism in the event of a fault or failure.

Each middleware product conveys these properties to a greater or lesser extent. The extent to which system transparency matches the needs of embedded systems will determine its usefulness.

2.2.2 Common Middleware

There are countless examples of middleware systems in use today. Each presents the user with some degree of transparency. Although all provide certain types of communications abstraction there are a number of ways in which they achieve this. Different systems use different metaphors for communication. These metaphors define the types of services that are provided and the way in which the users' programs interact with each other. Choosing a middleware system involves matching the interaction mechanism to the application; some tasks are more naturally expressed using a particular middleware. Other considerations, like platform support, and heritage may be important, depending on the applica-

tion. With patience, it's possible to mimic one metaphor with metaphor. At best this wastes otherwise productive time, at worst, it's possible to introduce errors [Lee, 2000]. Examples and explanations of common middleware systems are shown in Table 2.2. The next section considers in detail, those systems most appropriate for embedded space systems.

TABLE 2.2 Comparison of Common Middleware Systems

Metaphor	Explanation	Examples	Pros	Cons
Transactional	Transactions are conducted between clients and servers. Two-phase commit [Bernstein, et al, 1987] ensures integrity of databases, banking, etc.	IBM's CICS [Hudders, 1994], BEA's Tuxedo [Hall, 1996]	Very robust fault tolerance, concurrency control and scalability.	Narrow type of communication. High overhead, awkward implementation of arbitrary services.
Procedural	Remote clients can invoke specified procedures on host computer.	Sun's RPC [Sun, 1988]	Many supported platforms, very mature design.	Limited types of interaction. poor scaling and fault tolerance.
Message-Passing	Sender writes message and then sends to receiver. Messages queued at destination until asynchronous receive is executed.	Java Message Queue [Hapner, et al, 2001], PVM [Geist et al 1996], and many other variants	Flexible, conceptually simple, can be used to implement other systems.	Not very transparent, poor scaling, can lead to deadlock.
Distributed Shared Memory	Service provides illusion of large, shared memory space.	Berkeley's Network of Workstations [Anderson, et al, 1995], Brazos [Speight & Bennett, 1998]	Very intuitive approach for high-performance computing., efficient, fault tolerant.	Scales poorly, not very abstract, efficiency dependent on fairly predictable synchronization.
Publish-Subscribe	Servers advertise data services and update published values, clients subscribe to services to receive copies of values.	CORBA event service [OMG, 2001], RT-PS [Rajkumar, et al, 1995]	Simple, fault tolerant, fairly scalable and easily analyzed.	Constrained interaction mechanism.
Object	Clients may create, invoke, and reference remotely-implemented objects. Objects have persistence on server.	CORBA [OMG, 2000], DCOM, [Hortsmann & Kirtland, 1997]	Flexible, intuitive approach for many systems. Widely supported, powerful.	High overhead. Forces object-oriented approach.

2.3 Selected Prior Work

This thesis explores the development of real-time middleware for advanced space systems. Communications abstractions built into the software allow rapid-prototyping of high-fidelity simulations and help reduce the effort programmers must devote to managing software interconnections. Presently, few research programs exist that explicitly consider middleware for space applications, but parallels can be drawn from real-time middleware research and general efforts to improve flight software development.

2.3.1 Space Software Systems

Significant interest in software innovation exists within the scientific and small satellite communities. The former might adopt new technology as part of the mission definition, the later, because of budget limitations. While research into spacecraft autonomy is probably the most high-profile effort to improve flight software engineering, a number of other efforts address software development in general.

Mission Data Systems (MDS)

MDS is a research program at the Jet Propulsion Laboratory. Its initial definition identified thirteen software architecture *themes* [Dvorak, et al, 1999] that would define a common framework for flight software design and implementation. Examples of prominent MDS principles include: state-centric design, explicit use of models, goal-directed behaviour, and separation of data management from transport. MDS is closely tied to many of the autonomy research programs at JPL. We acknowledge that much of the initial inspiration for the GRRDE middleware system stemmed from the early conceptual definition of MDS. Our emphasis on explicit state specification and mobility (Section 4.1.2) is derived from MDS concepts. However, as the MDS concept matured [Dvorak, et al, 2000], it became less of a design approach, and more of a comprehensive product. Although useful for autonomy development, its reliance on goal-attainment and model-based behaviours suggested a substantial 'buy-in' threshold for designers. Our approach provides an incremental rather than revolutionary benefit, by facilitating rather than replacing current

design practice. So, while GRRDE shares some founding principles with MDS, but the focus of implementation is very different.

SuperMOCA

The Space Project Mission Operations Control Architecture (SuperMOCA) was developed at the Jet Propulsion Laboratory [Jones, et al, 1998]. SuperMOCA is not specifically described as a middleware, but many of the services the system provides serve the same purpose. The system is designed to provide abstract, self identifying interfaces between users and remote devices. Any device connected to the SuperMOCA network is encapsulated by direct I/O drivers and an I/O abstraction layer. Generic monitor and control mechanisms are then available to system operators, who can operate the spacecraft or instrument, without the need for device-specific training. The architecture is object based, includes a data-transport protocol and has provisions for transparent migration of functionality from ground to space. The status of the program goals is unclear, but the approach seems promising for high-level interaction.

Autonomy Testbed Environment (ATBE)

Also developed at JPL, ATBE is an approach to real-time simulation development for autonomous flight software systems [Biesiadecki, et al, 1997]. It integrates the simulation capabilities of the award-winning Dynamics Algorithms for Real-Time Simulation (DARTS) toolkits [Jain & Man, 1992], the DARTS-shell (Dshell), and component models into a common architectural framework for real-time simulation. The system is object-oriented and allows extensive visibility into the running simulation. Simulation elements are reconfigurable at run-time, which makes it a simple matter to inject faults or examine different operational approaches. ATBE has been used both in prototyping new missions and working around faults in existing ones. While the architecture interfaces with existing flight software, it is not designed to be included as part of the deployed flight software.

Object Agent

The Object Agent (OA) concept has been developed by researchers at Princeton Satellite Systems [Surka, et al, 2001]. OA is an agent-based approach to developing spacecraft flight software for distributed satellite systems. The authors describe a common framework for self-organization and flexible communication between software components. Each software agent obsesses a number of *skills* that execute concurrently and implement the functions of the agent module. Communication between agents is through natural-language messages. These messages contain data, addressing information, and time-stamps. They also contain a semantic identifier that specifies the structure, content, and intention of the message body. Simulations using OA software has demonstrated cluster formation-flying and collision avoidance algorithms.

2.3.2 Distributed Embedded Systems

Distributed implementations of embedded software are on the rise. Researchers see opportunities to apply the techniques of conventional distributed computing to advance the capabilities of physical devices [Bates, 1998]. Reconfigurable and composable systems are desirable goals [Kopetz, 2000], but experts acknowledge that transitions are not always easy. Temporal performance analysis is generally not composable [Lee, 2000] and local optimization can easily lead to mediocre global behaviour [Sha, et al, 1994]. In this section we examine several current approaches to real-time middleware.

Common Object Request Broker Architecture (CORBA)

CORBA is an open, distributed computing standard developed by the Object Management Group. As part of the general specification, it includes specific provision for real-time implementations [OMG, 2000]. CORBA is an object-based middleware service, designed to allow user client and server software to operate transparently with different, vendor-supplied transport mechanisms (i.e. the object request brokers or ORBs). So long as ORB implementations conform to the basic specifications, vendors are free to make different

design and domain optimizations. Space does not permit an exhaustive discussion of CORBA architecture, but many tutorials are available[1].

CORBA is probably the single most popular middleware system in use today, thanks to its extensive library of optional services and cross-platform support [Bates, 1998]. A good survey of Real-Time CORBA research is provided by Fay-Wolfe, *et al* [Fay-Wolfe, et al, 2000]. Several ORB implementations have been marketed as real-time, but many are still unsuited for hard real-time applications. Researchers at the University of Washington [Schmidt, et al, 1997] have designed *The ACE ORB (TAO)* from scratch to provide predictable and differentiated services in hard real-time environments. Testing results from this show promise of achieving acceptable real-time performance.

In addition to the basic ORB functions, the CORBA specifications also detail optional distributed services which ORB vendors may include with their products. The CORBA Event Service [OMG, 2001] specification adds extra client mechanisms to the basic ORB services. Normal interaction between client and server is through synchronous invocation. The event service allows asynchronous, 'push' type group communication similar to publish-subscribe systems [Emmerich, 2000]. Studies are underway examining the use of the CORBA event service with aeronautical flight control systems [Harrison, et al, 1997].

Simplex

The Simplex architecture was developed at the Software Engineering Institute (SEI) at Carnegie Mellon University [Seto, et al, 1998]. Simplex is a robust approach to reconfigurable control systems. The architecture allows designers to build modular control systems with variable levels of performance and analytic redundancy. Key components include I/O modules, control modules, switching logic, and the underlying communications system. Users provide several control modules, typically including some with low performance but high robustness. The switching logic is designed to exchange control-

1. *www.corba.org* is a good place to start.

laws 'on-the-fly' if the safety or stability of the system is at risk. This also allows the user to upgrade these control modules, while the system is running with guarantees of system stability. Researchers [Polze, et al, 2000] have integrated simplex components with CORBA to create tele-laboratory environments.

Supporting Simplex communications is a real-time publish-subscribe architecture developed by Rajkumar, et al [Rajkumar, et al, 1995], also from the Software Engineering Institute. It provides dependable real-time event-driven services between a network of processing nodes. Clients can publish or subscribe to named services. The designers provide enough transparency in the system operation to support conventional real-time analysis techniques.

2.4 The GRRDE Approach

We feel that the GRRDE approach is valuable, not just as useful middleware product, but as a demonstration of technology transfer between conventional embedded applications and the space industry. GRDDE improves the state of space-based middleware in a number of ways.

Information architectures for distributed space systems are only slowly being examined in methodical ways. Current plans to enhance software development, such as MDS, require significant risk and commitment from the systems developer. In traditional applications such as geostationary communications satellites the promised benefits may not outweigh the risks. In comparison, the services that GRRDE offers are both intuitive and can be scrutinized by normal software engineering methods. As programs move toward distributed computation, or simply want to reduce the programmers' ancillary workload, abstract communications services are valuable.

Both SuperMOCA and Object Agent offer communications abstractions to the software engineer. Although they address the distributed nature of emerging space systems, they do not deal directly with hard real-time concerns. These systems acknowledge the role of

hard real-time software, but consider it a localized, encapsulated part of the system. Consequently, the communication services are not presented with hard guarantees, or precise semantics. In contrast, GRRDE specifically targets distributed real-time systems. Our services are designed to aid development of the real-time software itself, and not just the high-level coordination.

CORBA and other object-based middleware, despite their popularity, may not be the best solution for all embedded applications. First, from an engineering perspective the overuse of object-oriented (OO) techniques has been criticized in high reliability systems [Hatton, 1998]. Furthermore, the functional metaphor of OO-design is usually more appropriate to conventional data-oriented software than function oriented real-time software [Wright & Williams, 1993]. Second, testing and validation can be difficult. As systems become increasingly abstract the fine-grain testing required for flight systems becomes difficult. This problem is exacerbated if the ORB source-code is not available. Third, CORBA adds significant overhead to systems that may not need all of its properties. GRRDE services are designed to produce low computational and memory overhead.

Simplex, is a perhaps the closest competitor to GRRDE and offers a similar collection of services. The differences between the two systems and the specific benefits of GRRDE are best understood after a careful examination of the GRRDE services and design. Therefore, we shall defer a comparison to this system until Chapter 11.

Chapter 3

THE GFLOPS TESTBED

This chapter presents the design of the simulator testbed known as the *Generalized FLight Operations Processing Simulator (GFLOPS).* Conceived as part of a study on *distributed satellite systems*, GFLOPS was to be a simulation testbed used to study the information processing requirements of collaborative satellite missions. Early analysis of missions such as TechSat 21 [Enright, et al, 1999] suggest that on-board processing capabilities will be crucial in systems that must digest large volumes of data. The original GFLOPS concept was to provide a generic platform with which to study information processing architectures for such missions. It was during the construction and early operation of the testbed that we identified the need for a system like GRRDE. In this chapter we examine the goals of the GFLOPS program, the construction of the testbed, and the evolution of the GRRDE middleware concept.

3.1 GFLOPS Formulation

The mandate of the GFLOPS program was to develop a software testbed suitable for simulating computationally intensive space missions. Its role was to facilitate software design and provide a feasible migration path from software simulation, through 'hardware-in-the-loop-testing', to deployed flight code. Balanced with the requirement for direct application was the desire to build a system capable of supporting not one, but an entire class of missions. Reconciling the requirements of generality and specificity appeared difficult.

The fundamental question was that of architecture; what would the testbed look like? Like any engineering process, this development required trade-offs. Accurate representation of certain capabilities precluded incorporation of others. The demands of traceability and accuracy suggested real-time software and embedded processors, while usability and flexibility considerations favoured other configurations. Moreover, the mechanism for the *simulation* side of the testbed operation had to be considered. Flight software would be useless without an environment with which to interact.

After consultation with industry and Air Force sponsors, the GFLOPS architecture was conceived. The simulator 'flight'-software would run in a real-time environment on processor families with space experience. System flexibility would be maintained, by selecting technologies that reflect projected future capabilities in space-qualified processors rather than current norms. Simulator tasks would be performed by conventional PCs. These PCs would also serve as the user's primary interface to running simulations.

Since our targeted missions involved distributed satellites, it was only natural that the simulation framework be inherently distributed as well. This provided three key benefits. First, flight-software and simulation software are physically separated. The distinct separation adheres to the "fly as you test, test as you fly" maxim, and removes the temptation to tailor flight software changes to the quirks of the simulator. It also helps to ensure that errors are not inadvertently introduced when moving from simulation to flight hardware.

Second, distribution would allow additional flexibility in representing different spacecraft bus structures. The range of architectures that could be represented include distributed processing *within* a spacecraft, tightly coupled collaboration for a cluster of spacecraft, or quasi-independent computation for a constellation.

Finally, 'transparent' distribution facilitates the migration of advanced software capabilities from the ground into space. Modules developed and scrutinized on the earthbound side of the command/data path, can later be moved onto the spacecraft without affecting the overall structure of data flow.

The following sections explore the details of the GFLOPS construction, its capabilities and limitations, and the emergence of the GRRDE middleware concept.

3.2 Testbed Anatomy

Although the GFLOPS testbed is primarily a software simulator, attempts were made in design to reflect the demands of the embedded computing environment. GFLOPS is founded on system processors chosen based on traceability to space-qualified hardware. Although many spacecraft still employ cyclic executives, such a solution did not grant us the flexibility that we required. Instead, we selected a modern, commercial real-time operating system (RTOS). Finally, software coding standards were developed to provide the user with guidance in simulation development.

3.2.1 Physical Architecture

The physical architecture of the testbed includes three primary elements: the mock spacecraft processors, support and simulation computers, and a high-speed network (Figure 3.1). All spacecraft flight software executes on the embedded processors and the simulation tasks are performed by conventional PCs. All of these components are connected by a high speed network.

The current hardware configuration for the GFLOPS system consists of both embedded and conventional computers as well as high-speed networking technology. Eight, 400MHz PowerPC750 processors were selected to represent advanced but attainable spacecraft processors. The computers are manufactured by Force Computing and at the time of purchase were considered high-end embedded processors. Each embedded computer is equipped with 256 MB of memory, 512KB of non-volatile memory (NVRAM), two serial ports and a 100Base-T network adapter. These processors use big-endian byte-ordering, an extra concern when moving data between the mock spacecraft and the little-endian ground computers. The processors are capable of a peak processing throughput of approximately 500 million floating point operations per second. The PPC750 is in the

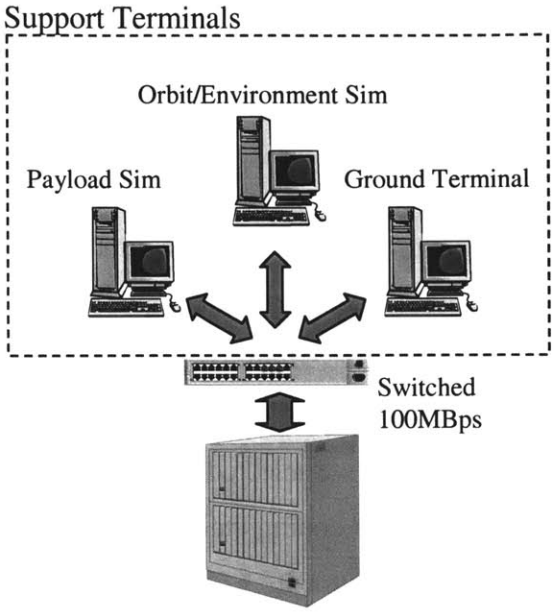


Figure 3.1 The Physical Testbed Architecture.

same processor family as the PPC603E, a design that is currently being radiation hardened [Brown, 2001].

The embedded computers are not attached to a mass storage device (e.g. a hard disk) although the operating system simulates such a device with a ram-disk. At present, the operating system and user applications are loaded from the network at system start. Such a configuration was deemed appropriate in a prototyping system such as GFLOPS.

Simulation tasks and ground station monitoring is performed by a number of conventional personal computers. These computers serve a dual purpose. First, they act as development workstations when the testbed is not in operation. Software development is accomplished with standard editing tools, and executable code is produced with the Green Hills MULTI cross-compiler. Second, during simulation operation, the PCs perform the various simulation tasks and act as an interface to the running simulation. The current configuration of the GFLOPS testbed employs three of these support computers. Typical task divisions are depicted in the above figure. One terminal acts as a user terminal, operating in context of

the simulation. The other two machines comprise the simulator. Environmental and 'engineering' simulation is performed on one computer, while the other replicates the payload operation.

Linking the computers is a local area network connected with 100Base-T ethernet. The network switch provides 100Mbps (duplex) capacity to each node. This network carries two types of data. The primary traffic is information transferred in the context of the simulation and carries inter-processor, inter-satellite, or satellite-to-ground messages. Secondary traffic represents connections to the simulators, that is, data collocated from sensors or destined for actuators.

Although this network design is not indicative of actual connection topologies or capacities in most systems, it was judged to be the best way to ensure testbed flexibility. The point-to-point throughput is comparable to some inter-satellite link capacities [Garrison, et al, 1995] and the selective connectivity or availability can always be accommodated in the flight-software design (Section 7.1).

3.2.2 Operating System

Historically, embedded systems such as spacecraft typically operate with a minimal operating system [Burns & Wellings, 1996], if they have any at all. Simple systems such as cyclic executives execute tasks in a fixed order and timing. While this approach is justifiable in deployed systems where processing and memory resources are highly constrained, it lacks the flexibility needed for rapid software prototyping. Additionally, management of the task sequencing can become quite burdensome for the developer as the project size grows [Verissimo & Rodrigues, 2001]. Consequently, we selected a robust modern operating system that provided hard real-time guarantees, but permitted more flexible computational models.

The RTOS selected was the OSE[1] operating system provided by ENEA Systems. The process model is based upon preemption and static priority assignments. Prioritized processes are designed to handle the bulk of system load, but other process types are also supported (i.e. timer or hardware interrupts, background). Flexible memory protection is offered and the designer can group related processes into common address spaces. OSE is a modern real-time OS that supports many features that make it particularly appealing for use in distributed systems. Inter-Process Communication (IPC) in OSE is achieved through message passing (termed *signals* in the OSE parlance). Furthermore, the IPC for OSE operates transparently across memory protection boundaries and through network links. The operating system also supports the capacity to load executable code at runtime. This facility is a boon to simulation debugging by permitting correction of small errors without the need for rebuilding the entire system.

Users of GFLOPS are assumed to have a working knowledge of OSE. The Enea website[2] provides high level information about the operating system, and the product manuals provide a good tutorial material. OSE is a modular operating system; developers may include only features that they require. Two methods of application development are supported. In *monolithic* kernels, the user's applications and the operating system are compiled together into one binary file. Processes may be created dynamically, but all executable code is loaded into memory at once. This approach is most efficient in its use of system memory. Alternately, OSE also supports the creation of *loadable modules*. The operating system must still be compiled into a binary file, and loaded at system start, but user applications can be compiled separately. The application modules can be stored externally and loaded into system memory (e.g. from network or hard drive) while the system is running. This approach reduces compile times for user applications, and allows flexible system configuration. Using loadable modules will incur some memory overhead since some executable code must be duplicated in each module, and the kernel must support the extra routines to

1. To the best of our knowledge OSE is not an acronym.
2. www.enea.com

handle loading and unloading the modules. However, it is a simple matter to use loadable modules for prototyping and early development, and then transition to a monolithic kernel later in the project. This approach is typical of GFLOPS development.

The following is a list of the commonly used OSE packages:

- Link Handler: Provides transparent network and serial connections to support message passing between processors. Users can write driver software to integrate OSE with non-standard network designs.
- Real-time clock: This is a coarse timing service. It generates notification 'alarms' at 1s granularity, but these can be set a long time in advance (e.g. days or more).
- Time-out server: This is a complimentary service to the real-time clock. The Time-out server provides notifications at 1 ms granularity, but are inefficient for very long intervals (several hours)
- OSE name-server: This package provides a distributed service registry for a network of computers running OSE kernels[1]. Processes may register services with the local name server. These services are identified by a text string. Other processes, either local or remote, may query the name-server to obtain the identity of the provider.
- Program Handler/Loader: This package manages loading and running executable modules. Modules can be loaded automatically by configuration scripts, or in an *ad hoc* fashion by the user.

There is a common dilemma in the development of embedded software. The more one commits to a particular operating system, the greater the range of tools and services the designer has to choose from; i.e. the less they have to develop themselves. Unfortunately these specialized features are often particular to the specific OS. Code or tools developed for one operating system will not transfer easily to another. Development of GFLOPS and GRRDE adopts an inelegant, but pragmatic solution. Since the number of software developers was limited, OSE capabilities were exploited whenever possible.

In defense of this approach, it can be argued that the promise of effortless embedded software portability is largely illusory. Even if corresponding functions exist, their temporal

1. It is not the same thing as a Domain Name Server

behaviour may differ requiring additional redesign or re-validation. Furthermore, when developer resources are constrained, one must make every effort to use available tools. Fully exploiting the capabilities of OSE provided the most effective return on our investment of time (i.e. it saved development time) and money (i.e. we had already paid for those extra capabilities).

3.2.3 Language Support

Choosing languages and coding conventions for real-time development is not always easy. This choice must balance traditional RT concerns of determinism, speed and space efficiency with more modern interests in readability, extensibility and expressiveness. On the other hand, object-oriented languages offer abstraction, encapsulation and the promise of reusability.

Despite the elegance that this strategy offers, the relatively risk-averse space community has typically been reluctant to adopt object-oriented programming on a large scale. Embedded systems frequently run into trouble when trying to employ source code not designed or particularly well suited for reuse. The failure of the maiden voyage of the Ariane 5 can be directly attributed to just such an occurrence (Lions, 1996).

The language selection philosophy of GLFOPS is to take an aggressive approach to adopting contemporary software engineering techniques, while retaining credibility with the spacecraft software community. To achieve these goals the suggested language convention is that of Embedded C++. EC++ is a subset of modern C++ that includes many of the important features of object-oriented programming (OOP) while acknowledging that the eventual target for the code is an embedded application. The restrictions on code content are driven by the following rationale:

1. Minimize memory usage and code size: Embedded processors typically possess only a fraction of the speed or memory of desktop computers. Furthermore for economic reasons it is desirable to maintain high processor loading (> 80%). Some C++ constructs and features can lead to significant inflation in both executable size and memory usage.

2. Maintain determinism: Timing and sequencing are critical to the successful operation of embedded systems. Wherever possible these guarantees must be maintained.
3. Promote a specification appropriate to the application: There are some features of C++ that do not truly contribute to run-time overhead, yet are not directly useful in embedded applications.

The following list describes the features of C++ that have been omitted from the EC++ specification:

- Mutable Specifier
- Exception Handling
- Runtime Type Identification
- Namespaces
- Templates
- Multiple and Virtual Inheritance

The interested reader is encouraged to refer to the official specification for further information [EC++, 2002].

Certain strictures are commonly enforced when developing software for space or embedded applications [e.g TRW, 1994]. The most striking of these is the prohibition against dynamic memory allocation. Generally speaking, the allowable behaviours are more structured and restricted. Determinism dominates flexibility. This need for determinism spans both time and space. Memory garbage collection can create poorly characterized delays, and running out of memory can be disastrous. Many RTOSs do not have heap management facilities at all. For similar reasons, the use of recursive algorithms are usually forbidden. Developers must maintain precise control over memory resources and processor access.

Some latitude can be permitted with regards to the strictness of these conventions. In many real scenarios it will be up to the lead software engineer to determine the limits of acceptable coding. Restrictions can be relaxed for some applications. For instance OSE

has an optional heap manager capability and EC++ has variants that allow templates and exception handling. Careful consideration should temper the use of such facilities.

3.2.4 Style and Naming Conventions

Many resources are available, providing tips and suggestions for good general programming practice. Some of the practices adopted for general programming are applicable to embedded systems. Other techniques must also be adopted to reflect particular idiosyncrasies of the real-time programming environment. Included in the appendix are several papers to this effect. One can summarize the suggestions into four general areas:

1. Process. These tips concern the practice of engineering entire software systems. When the system is complex and under concurrent development by many parties, steps are needed to ensure compatibility and ease of debugging.
2. Style. These suggestions relate to the physical appearance and structure of the code itself. Debugging one's own code may take an hour; debugging someone else's may take all day. Maintaining uniform and readable code can greatly increase the maintainability of software projects. While presently there is not an enforced style, adopting the suggested expression and commenting suggestions is encouraged. Regarding bracing it is suggested that users choose from either the KNR or One-True-Brace styles (Figure 3.2).

```
KNR Bracing:

if (foo (int a, char b))
{
        float baz;
        ...
}

One-True-Bracing:

if (foo(int a, char b)) {
        float baz;
        ...
}
```

Figure 3.2 The two suggested bracing styles. Either one is acceptable.

3. Variable Naming. Related to the previous point, conventions on naming variables, promote readable, maintainable code. For example SERTS naming convention [Stewart, 1999a] is an insightful specification. While not all existing code conforms to this standard, it is suggested that it be adopted for further projects. To further enhance readability the type-specific prefixes given in Table 3.1 are encouraged.

TABLE 3.1 Suggested Naming prefixes for GRRDE

Variable Type	Name Prefix	Example
int	i	iCount
unsigned int	n	nColumns
float	f	fSum
double	d	dInput
char * or string	s	sName
Custom Struct	st	stAddress
Custom Class	c	cMyObject
PROCESS[a]	pr	prLeader
bool	b	bActive
pointer	p	pCurrent
atomic object[b]	ao	aoAttitude
class/struct data member	m_?	m_iIndex

a. OSE Specific

b. GFLOPS Specific

3.3 Strengths and Limitations

All testbeds and simulations must make some pragmatic concessions to scope. Almost by definition, a testbed must abstract away certain details of the system that you are trying to represent so that you can explore the questions of interest. Thus, it is insightful to reflect on the capabilities of the GFLOPS architecture and to determine those features of space systems that it can represent well and those whose investigation would require another facility. Recognition of the limitations to a simulation technology are necessary to prevent situations where undue effort is expended to try and implement behaviours that the simu-

lator is ill-equipped to represent. Worse still, is a temptation to tailor a flight software solution to the peculiarities of the simulator rather than the real world environment.

The particular strengths of the GFLOPS architecture are best illustrated in comparison to some of its deficiencies. The least representative component of the GFLOPS systems is the network that connects the processors. Network capacity must be shared between contextual traffic and simulated data. The topology of the network is essentially a 'star'(Figure 3.3). There is nothing physically stopping any processor from talking directly to another member of the network regardless of the availability of such a connection in the real system. Furthermore, the latency and error rates of these communications links are qualitatively different than those likely to be encountered in a space system.

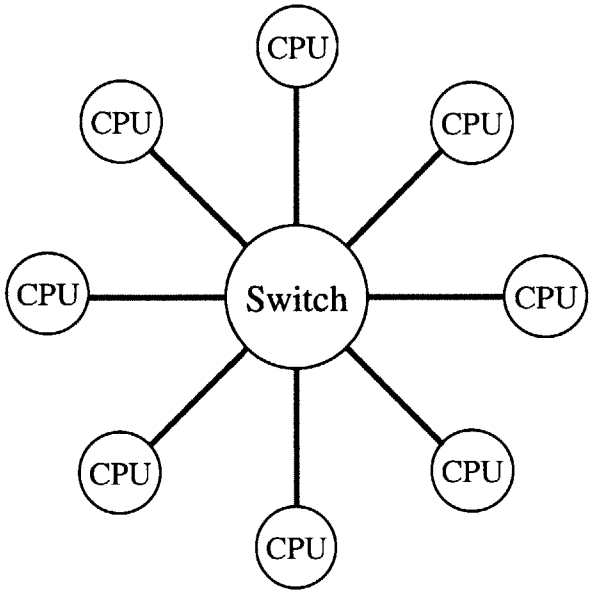


Figure 3.3 GFLOPS Network Configuration. Not all nodes are shown.

It is true that GFLOPS does not have the capability of easily representing these details, but these problems are of only minor concern from the point of view of the software designer. Careful design and abstraction limits the extent to which the developer must deal with these issues directly.

System limitations must sometimes be expressed directly in software. For example, since satellites can only communicate to the ground-station when it is in view, then it only makes sense for the satellite to *try* and communicate when it knows that the ground-station is visible. This knowledge can be conveyed to the flight software through simulated 'carrier-detect' sensors, for example. That the operating system provides certain connectivity, does not mean the user has to use the link until there is reason to believe that it is present. The same argument might be used when discussing link capacity. If the simulated mission has data pathways with less capacity than the GFLOPS network provides, it is incumbent on the developer to ensure that the flight software uses only the design bandwidth. Although the delivery timing may be different in GFLOPS, effective functional representation of low-capacity links is not outside the capabilities of GFLOPS.

To account for other discrepancies between GFLOPS networking and simulated communication systems, developers can use the same principles of abstraction that are employed in real designs. Distributed systems have long employed functional layering to hide certain communications details from the developer. For instance, the Open Systems Interconnect (OSI) [ISO/IEC, 1994] model allows web-page developers to ignore error correction, data routing, and a host of other details of the internet's internal operation. All of these functions are performed by low levels of software. The same holds true for a space system. Specialized communication hardware and low level software handle the framing, error checking, re-transmit requests, and other similar concerns. To the developer writing processing software, all they need to know is: which satellite they have links to, and the capacity that they are permitted on each link.

Another area in which GFLOPS does not provide a perfect simulation is directly at the hardware/software interface. Although GFLOPS tries to provide a platform capable of detailed, bit-level I/O, not all types of interactions are possible. Representing interrupt driven I/O is difficult. OSE allows the creation of software-triggered interrupts, but their responsiveness is not equivalent to hardware interrupts. Second, since the GFLOPS I/O interface is 'piped' through the network, sophisticated, time-dependent interactions with

external devices may not be feasible. Neither of these should cause much of a concern to the developer for two reasons. First, over-reliance on interrupt driven I/O represents a poor design in the first place [Stewart, 1999a]. Second, embedded software that provides a layer of I/O abstraction is generally regarded as being conducive to system reliability [Sha, et al, 1994]. GFLOPS can be used to simulate sensors and actuators at an abstraction layer. Hence, during system deployment, this abstraction layer must be replaced, but most of the remaining software can potentially remain intact.

Despite the difficulty in representing some details of system interfaces to hardware, GFLOPS can significantly aid the software development process in the middle to late stages of program development. As different elements of mission software migrate from offline prototypes to real-time environments, GFLOPS provides an excellent platform for performing embedded software engineering. Developers can map functions to processes, resolve task interactions and implement system timing functions and yet retain flexibility in the overall software configuration. Additionally, GFLOPS can support development when certain software modules are more evolved than others. For example, designers can begin implementing control algorithms with abstract implementations of sensors and actuators. As the development progresses and the interfaces mature, these details can be added to the simulation.

3.4 Emergence of the GRRDE Concept

During the definition and construction of the GFLOPS testbed, we realized that if GFLOPS was to deliver on its promises improving the software engineering process, we would have to develop a common framework for developing simulations. We assumed that users would adopt a modular development approach, but this alone would not be enough to significantly improve the software engineering process. Even with a common operating system and formalized programming conventions, software interactions would still need to be laboriously managed (Figure 3.4a).

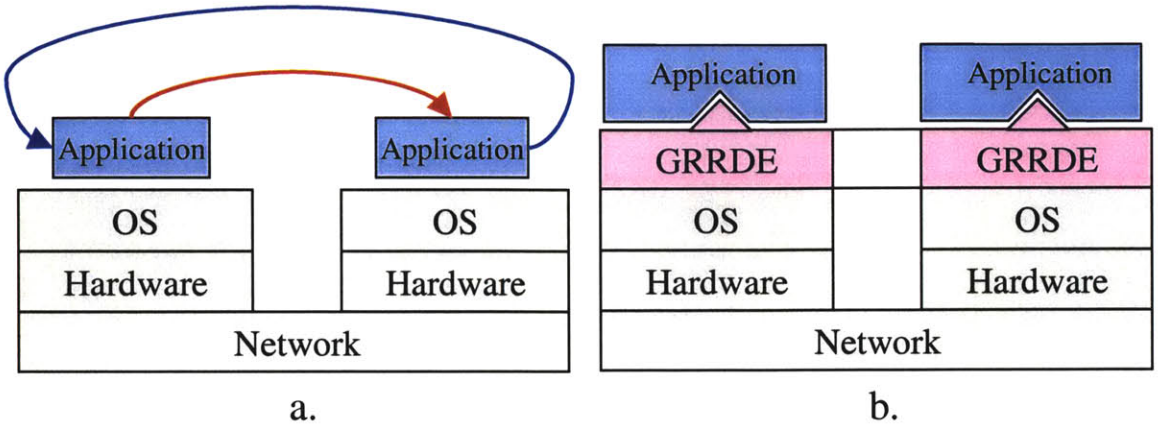


Figure 3.4 The Emergence of the GRRDE concept. On its own, each process must explicitly manage its connectivity (a). GRRDE provides a structured, abstract bridge between user software modules (b).

It was at this point that the GRRDE concept first emerged. OSE enabled transparent inter-processor message passing, but GRRDE would go a step further and provide powerful abstract communications services that would 'glue' a simulation together (Figure 3.4b). The challenge was to provide a common approach to developing simulations that would promote:

- Progressive Development. If each software module is relatively independent, this testing and development can be decoupled. Secondary system functionality can be added to simulations once key systems are proven.
- Rapid Reconfiguration. Different implementations of the same functional block can be exchanged so long as the external interface remains consistent.
- Functional Layering and Migration. Abstract monitoring and supervision can be introduced without affecting low-level software operation.

A middleware (Section 2.2) system seems to offer the best means of achieving transparent interoperability, but the constraints of developing software for embedded applications could not be overlooked. Some existing middleware offered real-time implementations, but the services they provided did not seem to be ideally matched to typical flight software design. Thus was the GRRDE concept conceived.

Chapter 4

THE DESIGN OF THE GRRDE REAL-TIME MIDDLEWARE

GFLOPS promotes efficient, reliable flight software development through a flexible and capable real-time simulation environment. Processing and memory constraints common to deployed flight software can conflict with the needs for instrumentation and reconfiguration during development. To address this problem, GFLOPS adds a minimally invasive service layer to the embedded operating system that promotes debugging and allows rapid, modular prototyping. This is the *GFLOPS Rapid Real-Time Development Environment (GRRDE)* middleware.

GRRDE provides publish-subscribe communications services to user supplied software modules. This strategy affords the user great flexibility in sub-function design and implementation while reducing the effort spent in managing software interfaces. This chapter presents the communications services offered by GRRDE. We begin with a discussion of modular software design in general. We then present a functional description of the two classes of publish-subscribe services that GRRDE offers. Finally, we consider the internal design of the software and examine the role that the user must play in constructing software modules.

4.1 Approaches to Software Design

The GRRDE middleware concept was conceived as a means of standardizing the approach to developing simulations with GFLOPS. The GRRDE run-time components

provide a glue that binds simulations together, but before it can be effective, flight software and simulators must exist. At the core of this design approach is modular functional decomposition. In this section, we present a summary of this common technique and the variations that we suggest for GRRDE-based development. We do not claim any particular innovation the theory of modular software design. Our changes represent shadings of philosophy adopted to make the resulting design more compatible with the GRRDE services.

4.1.1 Modular Software Design

The techniques known as *modular software design* have become so commonplace that they are rarely spoken of as such. They are simply called *software design* techniques. It is easy to forget that it was not too long ago that they were considered revolutionary ideas [Parnas, 1972] [Bergland, 1981]. Instead of belabouring what is essentially, common knowledge, this section first presents a quick overview of the essential features of modular design. We then discusses the near ubiquitous trend towards object oriented design and programming and finish with a consideration of how embedded applications influence the design process.

Modular software design refers simply to the structured and iterative decomposition of system function into identifiable sub-functions. Each sub-function should be responsible for a single task and have a clearly defined interface to other components [Bergland, 1981]. Effective design minimizes the interconnections between functional elements. Fewer couplings between modules creates systems that are less complex and less prone to error. Software structured in this manner can be implemented by separate teams, is easier to test, and is more suitable for reuse from one project to the next.

A complimentary effort to functional decomposition is the process of interface definition. Good software modules are black boxes; from the outside, the substitution or refinement of an internal algorithm should not have a direct effect on other components. They should *encapsulate* functionality. Not only is it necessary to document the data paths between modules, but the real-time nature of the application requires that temporal information be

specified as well. Careful attention should be paid to the units of measurement implicit in measured or calculated quantities. Temporal specifications typically include quantities such as task period, response time, and criticality. Explicit enumeration of all dependencies and interactions helps reduce the potential for *hidden interfaces*, known to be prime contributors to software-related failures [Leveson 1995].

Conventional software engineers commonly use object-oriented techniques to manage the complexity of large projects. These schemes provide a unifying metaphor for analysis, design and implementation that remains effective as the problem size grows. Object-oriented (OO) methods build on techniques of encapsulation and data abstraction developed for modular design. Instead of simply adopting these principles as conventions, OO programming languages implement these features directly in the language definition.

Several remarks must be made that relate specifically to design techniques appropriate for embedded systems. First, many traditional software applications are data-oriented, and strict OO analysis maintains that "objects are nouns". In contrast, real-time systems are generally oriented towards function more than data [Wright & Williams, 1993]. Thus, some of the more rigorous OO design techniques must be adapted for embedded systems. Second, embedded software engineering has been reluctant to embrace full fledged object-oriented programming (OOP) languages. It is worth noting that OO analysis and design can be beneficial, even without a full commitment to OOP. Implementation-neutral methods such as those proposed by Coad/Yourdan [Coad & Yourdan, 1991] are particularly suitable.

An example can provide insight into the modular design process. Consider the design of a controller for a simple water heating system (Figure 4.1). It should be noted that this is not a good design for such a system, it simply illustrates some principles of functional decomposition. The system must maintain the water level and temperature in a holding tank (*R*). Water is drawn into the tank by turning on the motorized pump M. Float sensors (*L1,L2*) indicate low and high water marks respectively. An unspecified and time-varying dis-

charge (Q) removes water from the tank. The controller must regulate the temperature of the water with the aid of a variable power heating element (P), and a temperature sensor (T). Lastly, the controller must be aware of the potential danger of low water level and high temperature. This represents an alarm monitor (A).

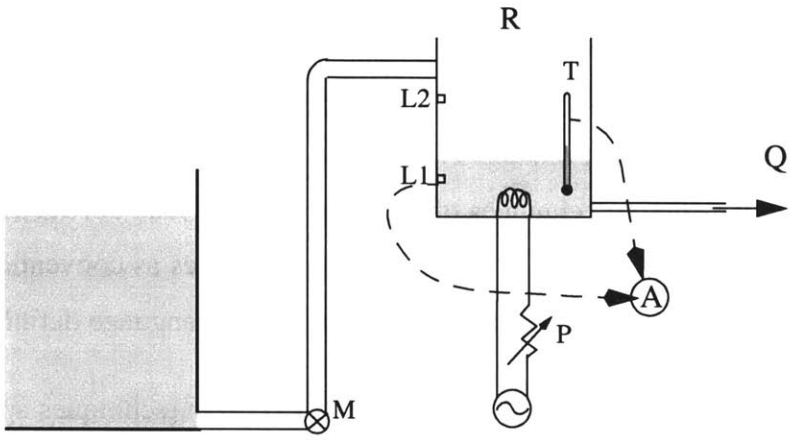


Figure 4.1 A simple control system example.

A possible decomposed design for this systems is shown in Figure 4.2. The level estimator performs some simple filtering on the binary data from the float sensors and outputs a qualitative estimate of water level (e.g. high, nominal, low). The motor controller turns the pump on and off based on the level of water in the tank. In a practical system, this may involve a particular start-up sequence for the motor that isn't captured in the simple figure (Figure 4.1). The temperature controller adjusts the power output to the heater to maintain the reservoir temperature at a constant level. Overseeing these two modules is an alarm monitor, whose job is to check for potential safety violations and issue override commands to the low level controllers if anomalous conditions are detected (e.g. automatically shut off heater if water level is too low). While the design solution to this example is fairly trivial, it does illustrate principles of modular functional allocation.

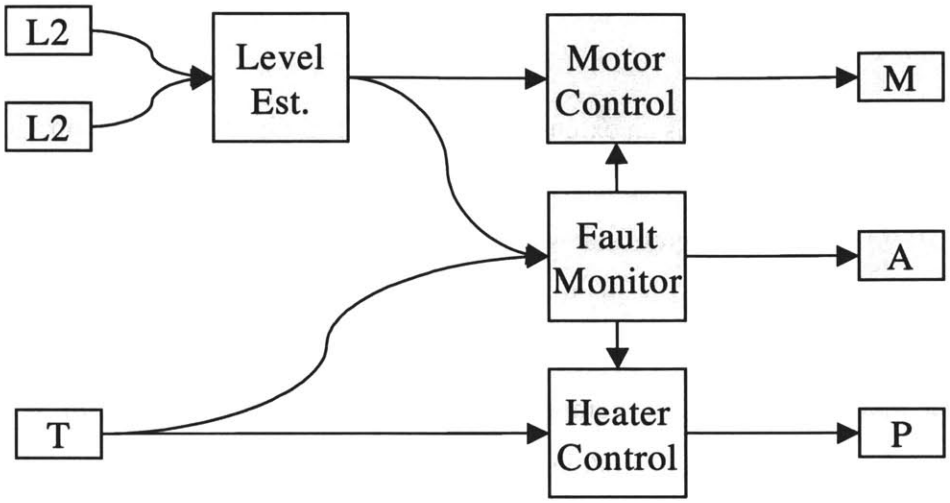


Figure 4.2 Functional decomposition for water heater example.

If GRRDE is to appeal to the embedded community, and flight software developers in particular, we must acknowledge the role of any existing design culture. Companies developing software for spacecraft will have their own heritage of design strategies and conventions. Methodologies requiring drastic changes to the users' design approach are difficult to sell when reliability and predictability are important. Consequently, GRRDE attempts to be minimally prescriptive in regulating the user's approach. GRRDE does require a commitment to modular software design. Without clearly defined interfaces, and segmented software functionality, the benefits of GRRDE communications services are lost. Although we have used the Embedded C++ language (Section 3.2.3) in developing the GRRDE run-time components, user applications need not be object-oriented. Only one other restriction is made on design. GRRDE is most effective when combined with design that emphasizes the flow of state information.

4.1.2 State Centric Design

From the design process, specifications can be developed for each functional block. The specification process describes the abstract function of each module (including timing information), its inputs, its outputs, and lastly, any external dependencies. A central tenet

of the GRRDE design process is to think of these features in terms of state information (or simply state).

The GRRDE concept of state is an extension of the concepts discussed in the previous section. Traditional state machine notations used in Object Oriented Analysis and Design [Costain, 1995] deal with discrete states and transitions. One can think of this in terms of the mode an object or system is in. In embedded applications it is necessary to reason about continuous quantities as well. Spacecraft have continuous orbital elements, a chemical reactor may have an internal temperature, etc. Defined generally, a *state* is a set of internal variables that reflect a module's perception of some aspect of the system. The level of state abstraction varies greatly depending on the particular application. Thus, "The status register of the star-tracker contains 0xffec0001," represents a possible state as is, "The spacecraft altitude is 500.021 km," or "The +Z reaction wheel is acting erratically." The level of abstraction can be adapted to the maturity of the design or the degree of abstraction at which a particular software component operates.

Blocks cannot operate completely independently. Inputs and outputs identified in the design process should be expressed in terms of state as well. Inputs represent the state information that the block requires. This interface can either be directly to hardware (i.e. a sensor reading) or to another software module. Outputs are likewise a specification for a certain type of information that a module can provide. The data flow through a system can be charted by identifying the sources (i.e. providers) and sinks (i.e. consumers) of state information.

Using this emphasis on state information, a block's function can be viewed as operations on its state (Figure 4.3). In this formulation, each functional module will consist of input and output states, optional internal states, and logic that provides transformations between them. Common functions of embedded software include:

- Signal Conditioning. I/O interfaces often involve elements of signal processing. Incoming sensor data is frequently filtered or combined with other sensors to provide a synthesized estimate of a more useful quantity. Similarly,

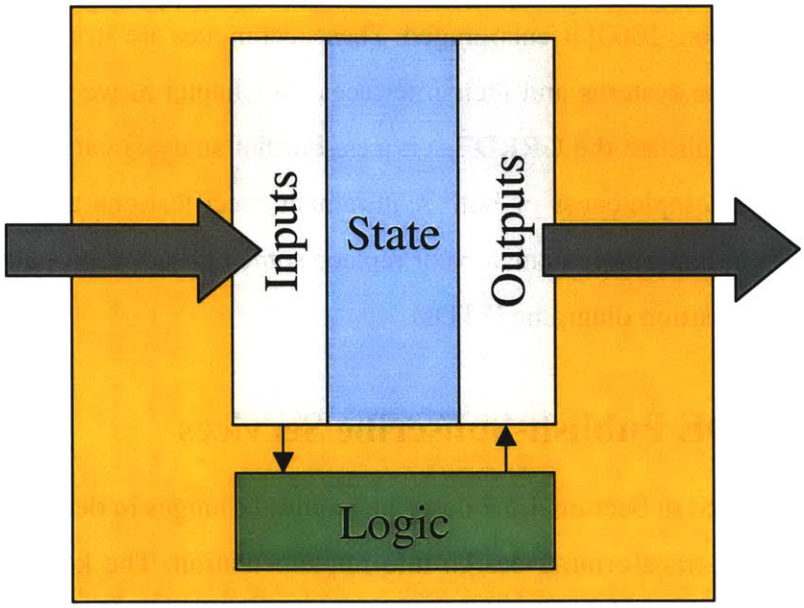


Figure 4.3 Viewing a module's functions as state transformation.

discrete actuators can be used to provide analog-like performance when pulse-modulated.

- Control Systems. Digital control systems are features common to many spacecraft. These feedback driven systems can be used to maintain attitude, steer antennas and solar arrays, suppress vibration, charge batteries, and regulate temperature. Control systems must take command inputs, and sensor readings and produce appropriate output commands to system actuators. All of these quantities express types of state information
- Monitoring and Sequencing. Elaborate systems have complex control structures. Different control relations typically hold when devices are in a steady state than when they are maneuvering dynamically. Functional modules may have discrete states that track the current internal mode of the device. High level modules can track these status indicators as a means of coordinating elaborate behaviours.

Although the GRRDE concept of state centric design is quite flexible, it does not capture all the allowable interactions between modules. Designers are free to make use of direct interaction between modules or processes if the application warrants it.

Adequate representation of the full range of system characteristics may require additional tools. To aid in the specification and design of GRRDE-based systems, the use of a formal

specification standard such as *Input/Output Automata (IOA)* modelling [Lynch, 1996] or SpecTRM-RL [Leveson, 2000] is encouraged. These techniques are structured formalisms to specify and analyze systems and their interfaces. In Chapter 5, we actually employ a variation of IOA to validate the GRRDE services. Formal analysis can be used to verify the correctness and completeness of both requirement specifications as well as detailed designs. These specifications can enhance or replace some of the traditional OOAD products such as state transition diagrams (STDs).

4.2 The GRRDE Publish-Subscribe Services

The concepts discussed in Section 4.1.2 describe required changes in design emphasis, but offer little insight on transforming design into implementation. The key innovation that GRRDE provides is a set of communications services designed to facilitate rapid and robust development. This section presents a functional overview of these tools and illustrates their application to simulation design.

4.2.1 The Subscription Concept

In the previous section, we examined the role of state information in the GRRDE approach to modular simulation design. Simply identifying module interfaces does not, in itself, make development much easier, or a design more extensible. Middleware adds value to a system by introducing transparency (Section 2.2.1) and abstraction in the interfaces between components. Consider the simple system illustrated in Figure 4.4. We have a sim-

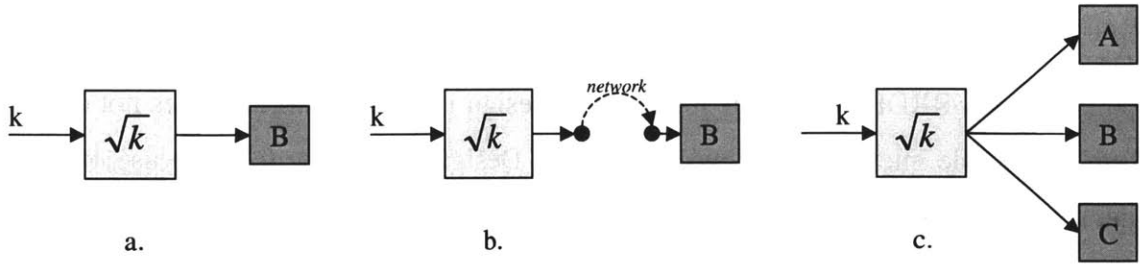


Figure 4.4 Key dimensions of GRRDE transparency. Modules in a simple system (a), may be moved to remote processors (b), or connected to additional monitors (c).

ple module that takes the input k and computes $\sqrt{k}$. In our initial design (Figure 4.4a), we have one module B, that needs to use the value $\sqrt{k}$. What happens when design revisions force the module B to a different processor (Figure 4.4b)? or when enhanced functionality dictates that modules A and C also need access to $\sqrt{k}$ (Figure 4.4c)? In the GFLOPS system, the first scenario is adequately handled by the OSE link-handler, the OS' transparent messaging service. The second is a little more complex.

The question that arises is one of responsibility. Which module is responsible for the data pathways shown in Figure 4.4b (or c)? Is it the duty of the server module? This solution is troublesome, since it appears to break the encapsulation of the square-root module. Changing the internal software each time another destination is added or removed seems laborious and error-prone. On the other hand, the burden of maintaining the link may lie with the client modules. If this is the case, then they will need a means to access the value of $\sqrt{k}$. This solution seems promising, but some connectivity issues remain. These are:

- Persistence. Data connections in an embedded system are usually permanent. *Ad hoc* information access is not unknown, but not usually of primary concern. We would like to store enough persistent information about the link so that regular state access to $\sqrt{k}$ has low overhead.
- Communications Metaphor. This is related to the previous point. Data access is frequently repetitive. We would like ways to simplify data access mechanisms in the client software.
- Data Access. Exactly how are data made available to the clients? Shared memory offers some potential, but is only available for local processes. In addition these solutions can constrain how modules are implemented. For example a shared memory solution requires calculation of $\sqrt{k}$ at least once for every new value of k. However, in some situations, we may want to calculate $\sqrt{k}$ only when a client wants to know about the value.

The GRRDE publish-subscribe are services used to automate the delivery of state information. Servers tell GRRDE about the types of state information that they *publish* and how to access the information. Clients can then request subscription *contracts* for a particular data service, subject to certain constraints. GRRDE will then collect and disseminate the information based on the parameters of the contract. The GRRDE publish-subscribe

system (GPSS) allows subscribers of state information to receive updates of state information without disturbing the operation of the publisher. These features decouple the process of operating on state information from the act of distributing it. Thus, the logic of a module can be written without reference to the precise origin or destination of the state information.

Cyclic low-level processing is common to embedded systems. Many use periodic digital controllers that monitor and control a physical process or other quantity. Higher level functions, such as those that change operating modes, are aperiodic. Both schemes rely on the movement of structured information, be it a velocity measurement, a status report or a command directive. To maintain modularity and promote system reconfiguration, the GPSS allows the information pathways to be configured dynamically. Sources and sinks can be addressed abstractly, and additional data flow can be added with minimal disturbance. This creates three main benefits:

- Progressive Development. The relative independence of each module allows subsystem testing and development. Secondary system functionality can be added to simulations once key systems are proven.
- Rapid Reconfiguration. Different implementations of the same functional block can be exchanged so long as the external interface remains consistent.
- Functional Layering and Migration. Abstract monitoring and supervision can be introduced without affecting low-level control.

The OSE operating system provides a distributed service registry (Figure 4.5). Operating across multiple processors, a process or block[1] may register one or more named services (e.g. text strings) with the registry. These services typically describe some function of the source block. For instance, a block may register itself as an "Ephemeris-propagator." Other processes query the registry for services that they require. If matching entries exist, the registry service will provide the querying process with an identifier with which to contact the service provider. This service name must be part of the module specification. The interface documentation for each block must also specify the data products that it can pro-

1. A *block* is a collection of processes in the OSE operating system

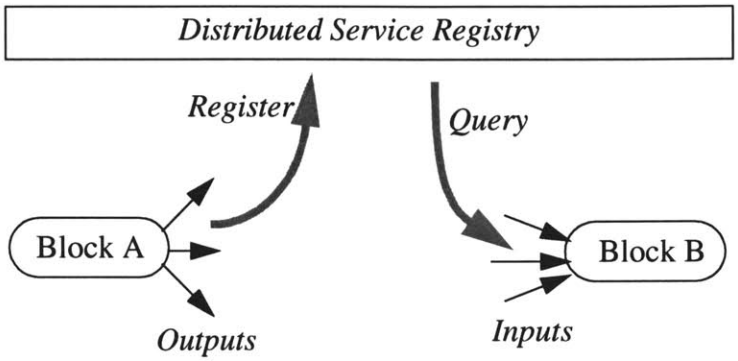


Figure 4.5 The OSE Nameserver. State producers register output types with registry service. State consumers query registry to locate producer.

vide. This specification must include a label for the type of data as well as a detailed description of the data format. The latter may include actual data type-declarations. Some simple examples are given in Table 4.1.

The role of the output specification is quite straightforward. If a block publishes certain data products then any external process may obtain the published data products by invoking the GPSS. Upon initialization, the source block provides the GPSS with access routines to obtain the pertinent information. The GPSS will pack the data into a formatted signal and send it to the destination process when requested. The destination process or block must be prepared to accept and interpret the contents of the message. This mechanism is detailed in Section 4.3.

Two important restrictions affect the interface definition process. In any system configuration, all named services should have a unique provider. For example, no two modules should be named "Attitude Estimator." Furthermore, there is no formal semantic structure to the service identifiers. The identifier "position_of_spacecraft_1" has no relation to the identifier "position_of_spacecraft_2" as far as the service registry is concerned. Although it would be theoretically possible to implement software to remove these restrictions and allow more flexible information brokering, such service has been deliberately avoided. In an embedded system any such arbitration should be handled explicitly by the user. Deter-

TABLE 4.1 Sample Block Output Specifications

Service	Data Product	Format

Ephemeris	Orbital_Elements	structure OrbEls
		{
		double Time (seconds)
		double a (metres)
		double e (unit-less)
		double i (degrees)
		double RAAN (degrees)
		double omega (degrees)
		double MnAnom (degrees)
		};

	Position_Velocity	structure PosVel
		{
		double Time(s)
		double X (m)
		double Y (m)
		double Z (m)
		double Xdot (m/s)
		double Ydot (m/s)
		double Zdot (m/s)
		}

Momentum Wheel Control	Torque_Output	double Torque (Nm)

minism must take priority over flexibility. Consequently, modules needing to receive inputs in this manner must have knowledge of the service name, data product name and message format.

The GPSS brokers agreements between source and sink for repeated delivery of a certain data product. The resulting agreement is called a *subscription contract*. Contracts can take one of two forms: periodic and aperiodic. Periodic contracts send regular updates of the data at fixed intervals. They may run indefinitely or for a specified amount of time. Continuous variables are typically represented with periodic contracts. Aperiodic contracts, by comparison, will send updates only when the state value changes. Discrete states such as mode or status are ideally captured by an aperiodic contract. Periodic and aperiodic con-

tracts can also be referred to as *time-triggered* and *change-triggered* contracts, respectively. The type of contract that a given data product will support must appear in the specification. The GPSS does not allow one data product to support both types of contracts. This was a necessary compromise to ensure efficient message dispatching and to restrict unnecessary message traffic. These two varieties of subscription are examined in Section 4.2.2 and Section 4.2.3.

Not all inter-process communication requires message contracts. Operations such as issuing commands or handshaking are best performed directly through direct signal (message) exchange. It should be noted that complicated protocols are more susceptible to logical design errors and may be difficult to characterize temporally. Designers are therefore encouraged to use the GPSS whenever possible.

This state delivery mechanism makes the systems developed with GRRDE inherently modular. The information paths (i.e. state delivery) can be configured at runtime, eliminating the need to break encapsulation to make the appropriate connections. Functionally identical blocks can be interchanged if they both provide the same public state outputs. The resulting modularity promotes rapid and effective simulation development through the complete development process.

4.2.2 Time-Triggered (Periodic) Subscriptions

Time-triggered contracts are most commonly encountered in low-level control systems. Situations where information updates are expected at regular intervals suggest the use of this type of contract. Specification of the service will include the update rate of the publishing module. Subscribers typically request a time-triggered contract at this rate or slower. Faster rate contracts are not prohibited, but are unlikely to be effective since they lead to duplicate values being delivered to the clients. Users must also be aware of the potential for under-sampling time domain phenomena when slow sampling rates are used. Synchronization issues are discussed in Section 7.2.3.

Possible uses for time triggered contracts include:

- Sensor Polling
- Control Actuator Commanding
- Telemetry Logging
- Watchdog Services

4.2.3 Change-Triggered (Aperiodic) Subscriptions

Change triggered contracts are useful for data abstraction and status indication. For example, a sequencer may give the attitude control system (ACS) a command to change orientation. The *ACSstatus* may then indicate values of *slewing* and then *CoarsePointing* and finally *FinePointing* as the system stabilizes. Subscribers receive an updated copy of the subscribed variable whenever the value is changed. In essence, this type of service is a primitive form of multicast (one-to-many) communication. These subscriptions are useful for communicating with supervisory modules as well as peers.

Possible uses include:

- Qualitative status or health
- Operating mode
- Command feedback
- Multicast

4.3 GRRDE Internals

We have discussed the services that the GRRDE middleware provides. Periodic and aperiodic subscriptions allow a module to publish state information for general consumption. In this section, we examine the composition of the GRRDE system and consider how subscriptions are initiated and filled. The most important parts of the middleware are the mechanisms that allow a module to publish and distribute its data. We also discuss contracts from the subscriber's perspective and consider how incoming messages are handled. We conclude with some general observations about the middleware architecture.

4.3.1 Dispatch Functions

GRRDE relieves the user from managing incoming data requests or contract subscriptions. If the middleware is to take on this responsibility, the user must tell GRRDE how to find the information it needs. To publish a particular type of data the user must register a *Dispatch Function* with the GPSS and associate it with a *Data Product Name.* Dispatch functions must allocate signal storage, fill the signal with appropriate state information and send it to the destination process provided in its calling parameters. One dispatch function is typically required for each data product name that a module provides. A limited parameterization of the function's behaviour allows a measure of flexibility in providing slightly different responses to different contracts.

Upon initialization, a block must register the dispatch functions along with their identifying text labels. This information is placed into a table which associates a numerical index with the text name and the memory address of the dispatch function code. This table is used by two processes that manage contracts and block outputs. The *Message Negotiator*, a low-priority process, receives requests for new contracts and records the details (e.g. period, destination, etc.) in a second table. Each contract is associated with an entry in the dispatch function table.

The high-priority *Message Dispatcher* function invokes the dispatch functions associated with active contracts. The dispatch of periodic subscriptions is triggered by a system timer. Aperiodic contracts are triggered when their constituent variables are changed. The operation of the dispatch mechanism is depicted in Figure 4.6.

A dispatch function for a periodic publication is straightforward to implement. Its only responsibility is to generate the data signal and send it to its destination. The signal contents are typically read from a shared variable inside the software block. The logic processes update this variable periodically, and the dispatch function reads the current value and sends it to the subscribers. Special Embedded C++ classes called atomic objects guarantee exclusive access to the data during read and write operations. Dispatch functions

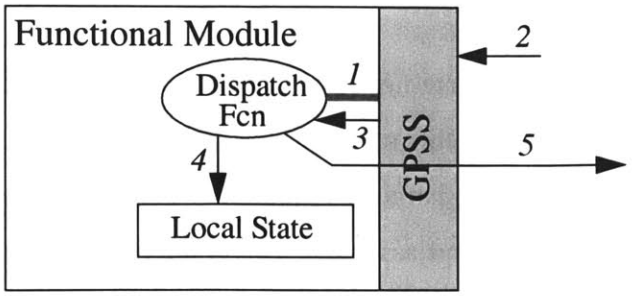


Figure 4.6 GPSS Operation. On initialization (*1*), module registers dispatch function with GPSS. External data or contract requests (*2*) trigger function (*3*) which reads (*4*) and transmits state information (*5*).

may perform calculations or access multiple variables, but optimal performance is achieved when they are as simple as possible

Aperiodic contracts are substantially more complicated. Let us first consider how the dispatch signals are generated for these contracts. Associated with each named data service is one or more *atomic2* objects. These are implemented as classes derived from the simple atomic objects described above. In addition to providing safe, concurrent read and write access to their contents, *atomic2* objects generate dispatcher notifications. Consider the dispatch sequence shown in Figure 4.7. When the module's logic process writes a new value to the *atomic2* object, a trigger signal is sent to the message dispatcher. The dispatcher then invokes the dispatch functions for all contracts that reference that object. These dispatch functions must read the atomic variable, and send off the appropriate signal to the subscriber.

One *atomic2* object may be referenced in several dispatch functions, but each dispatch function may access more than one *atomic2* object. We must ensure that each contract that references the *atomic2* object is dispatched exactly once. The dispatcher does not know the details of these dependencies, and so it attempts to invoke every active aperiodic contract. It is the responsibility of the dispatch function to decide whether to send out the dispatch signal.

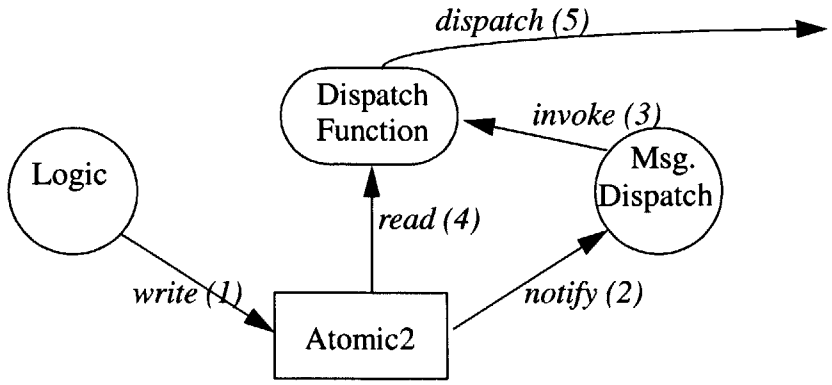


Figure 4.7 Dispatching an aperiodic contract. Numbers indicate the execution sequence.

Associated with every contract is an unsigned integer tag. This tag is passed to the dispatch function from the message-dispatcher. Each *atomic2* object contains a flag that is toggled between '0' and '1' every time a new value is written. When invoked, the dispatch function checks the flags on each *atomic2* object that it references, and concatenates the flags into a single binary value[1]. This number represents the new tag for the current contract. If it matches the old tag, the dispatch function should not send out a publish message. A different tag value indicates that at least one of the dependant variables has changed, and a publish message should be sent to the subscriber. The new tag value is stored with the contract record.

This strategy complicates the process of writing dispatch functions, but carries several benefits. All the knowledge about data dependency is encapsulated within the dispatch function. The functions may be complex, but the complexity is purely local. Moreover, we have developed convenient functions that automate the process of tag generation. The user simply provides a list of the component atomic objects. Since the flag evaluation order is the same every time the function is called, the tag values are consistent. Using this type of

1. Thus, the current configurations limits each type of subscription to depend on a maximum of 32 *atomic2* objects. This was judged to be sufficient for most systems.

structured interaction means the message-dispatcher does not need to know any details about the data product composition.

4.3.2 The Input Arbiter

Signal inputs to a module are less automated than output publications. During initialization, a module will create a dummy or *phantom* process that acts as the address for all incoming communication. OSE allows this process to contain no executable code, just redirection information for incoming signals. When a module registers a named service, it presents this dummy block as its address. This addressing mechanism hides internal implementation details from the outside world. The GPSS is able to catch any of its own messages such as contract requests at this level. Remaining incoming messages are redirected automatically to a user-supplied *input arbiter* process.

The input arbiter's job is to copy the incoming messages into local state storage, or redirect them to the appropriate internal process. Frequently, these processes consist of nothing more than a check for incoming signals and a large 'switch' statement in a perpetual loop. The message tag is checked against the supported values, and the incoming data is routed to the appropriate location. Atomic objects are used to manage concurrent access between the arbiter process and internal module logic. The input arbiter logic can be used to implement synchronization between software modules (see Section 7.2.3).

4.3.3 The GRRDE Middleware Architecture

The GRRDE middleware adds several processes to each user module. The Message Dispatcher and Message Negotiator handle outgoing publications and the Input Arbiter directs incoming traffic. Although we use the OSE NameServer as a distributed service registry, the core GRRDE components are essentially localized and do not interact with one another. Each module handles all of its own outgoing communication and relies on the message-passing built into the operating system. Once signals leave a module, they are delivered like any other OSE message. Some of our synchronization tools implement dis-

tributed algorithms, but the essential GRRDE services are self-contained. This has the following ramifications:

- GRRDE is fault tolerant. Since communication is handled locally, the effects of a failure of a process, a processor, or a communications link are only seen locally.
- GRRDE operations are easily monitored. Since GRRDE exploits the native communications services, existing profiling and debugging tools can be used directly.
- GRRDE is interoperable. Since GRRDE uses native message-passing, GRRDE modules can publish data to non-GRRDE clients.
- GRRDE Communication is most efficient in hierarchical software designs. Since the internal GRRDE processes do not coordinate their message delivery efforts, multiple remote recipients of the same data product will generate extra network traffic. Consequently, we discourage *ad hoc*, peer-to-peer subscriptions. Instead, we encourage system designs that explicitly recognize inter-processor coordination as a distinct type of software function (e.g. a module that has the specific task of estimating cluster geometry, by collecting position data from each satellite).

This last point, in particular, may be viewed as a disadvantage, but we believe that the overall structure of the middleware is consistent with the needs of embedded space applications.

4.4 Interface Definition

We have discussed how GRRDE is founded upon the principle of information mobility. Section 4.2 detailed how a formalized input/output design can be converted to a corresponding modular implementation that decouples the generation of state information from its delivery. This section details the contents of the interface specifications. Each entry in the interface specification must describe an input or an output. When a system is composed of several modules, the inputs to all blocks should appear in the output specification of others. The run-time GRRDE components then form the appropriate data pathways.

4.4.1 Interface Classification

These inputs and outputs can be labeled in terms of the type of connection that they assume. Many types of inter-module communication can be described by one of three types of exchanges; these are: time-triggered, change triggered, and command. The first two categories refer to I/O that can support subscriptions. Time-triggered contracts are updated at regular intervals and usually carry pseudo-continuous state information such as position. Thus, a time-triggered output is a specification that a given module will make a certain data product available at a given rate. Input specifications document instances that a module will request that a particular type of data is delivered at a specified rate. It is permissible that the input rate be slower than the rate specified in the output specification of the source block. Change-triggered contracts typically correspond to discrete states such as an operational mode. They are updated aperiodically, and subscribers are only notified when the value changes. Command inputs and outputs are also aperiodic but differ in philosophy. In contrast to a change-triggered contract, commands usually carry imperative content. Thus, a formalized command input may specify that the Attitude Control System module can accept SLEW_TO directives. Command relations, coupled with change-triggered monitors represent a useful architectural element. For example a supervisor module may issue a MOVE_TO_XY command to a control module and then monitor the subordinate's mode to check for completion (i.e. the mode changes from MOVING to IDLE).

Realistically, we acknowledge that there may be situations where the above taxonomy does not adapt well. In fact, Chapter 10 introduces two types of communication not covered by this classification Specialized communication is permitted but should be employed with care, since the user must manage all the details of the interaction.

4.4.2 The GRRDE Interface Specification

Successful integration of disparate modules depends upon effective management of the interface documentation. The specification process requires several steps. First, system functions should be decomposed and allocated to modules. This outlines the major func-

tional units of the software and their scopes. A second phase examines each module to determine the data that it requires to operate. The identified types of information are grouped with related quantities. In the third phase, the specification is formalized with emphasis on module *outputs*. Modules become responsible for providing particular types of information. We allow a single source to provide data to multiple sinks, but prohibits the converse. For example, many modules might use POSITION but only one is allowed to provide POSITION. Thus, the specification of an output element defines a new type of data service, but the specification of an input element is made only in reference to a previously defined output. The implications of this distinction will be revisited after describing the specification.

The following items must be specified for each interface element.

Service Name and Data-Product Name

The two primary characteristics of an output or source element are the *service name (SN)* and *data-product name (DPN)*. These two categories enforce a loose structure on a system's interfaces. A sink module locates a source module by querying the operating system with the SN. If the service is found, the sink may then contact the source to access the interface element identified with the DPN. For change- and time-triggered interfaces, the semantics of the DPN are directly meaningful since the contract setup mechanism uses the DPN as an identifying character string. DPNs used in command specifications are meaningful only during design and are not used by the GRRDE run-time components.

GRRDE allows non-unary mapping between SNs and DPNs. First, several DPNs may be associated with a single SN. For example, Module-A may provide a service with SN = MOTION_ESTIMATE, associated with two output interface elements: DPN_1 = POSITION and DPN_2 = VELOCITY. Second, GRRDE allows multiple SNs to provide the same set of DPNs. This property can be used to identify specific copies of duplicated modules in a distributed system. For instance, a group of networked aircraft may share common software. To avoid confusion, the SNs in each aircraft are appended

with a unique identifier (e.g. AUTOPILOT_1, AUTOPILOT_2, etc.), yet they share the same set of DPNs. Such SNs are denoted SN = BASE_NAME_UID#.

It is important to note that the association between SN and DPN is purely a design-time formalism. The SN functions as a lookup mechanism to find a particular module. The DPN is used when querying a particular module for its data products. If MODULE-A's specification lists SN-1 associated with DPN-1 and SN-2 associated with DPN-2, there is no run-time protection against finding the module with an "SN-1" query and then using the result to obtain the DPN-2 data (rather than doing a second query for "SN-2"). Designers must be careful to avoid such shortcuts since any redistribution of functionality between modules may render the initialization logic invalid.

Element Type

This specification parameter simply identifies the type of interface element (Time-Triggered, Change-Triggered, Command, or Other) and whether it is an input or output. Most interfaces will use one of the three basic types. Elements of type 'Other' require supplemental documentation of the interface operation.

Signal Number (SigNo)

The signal number is the OSE identification tag that labels the inter-process signals. Each interface element should have a unique SigNo.

Structure

Since GRRDE inter-process communication uses the OSE signal interface, it is important to describe the information content of each type of signal. Each signal must be referenced to a C-language structure definition. Two acceptable formats are:

```
typedef struct {
double X;
double Y;
} my_data_type;

typedef struct {
```

```
SIGSELECT SigNo;
my_data_type Data;
} my_signal_type;
```

or simply,

```
typedef struct {
SIGSELECT SigNo;
double X;
double Y;
} my_combined_type;
```

Both of the styles presented above have a SIGSELECT definition as the first entry in the structure used directly as a signal. It is a numeric tag that identifies the type of message. The first format separates the declarations into a signal declaration, and a 'payload' declaration. It is a convenient notation when the entire contents of the signal will be moved or used at once. The second format employs only one structure. It results in notationally compact code (fewer levels of indirection) when parts of the data are examined or entered separately.

The following list addresses some stylistic concerns:

- Both structure definitions do not require the same amount of storage. Because of byte ordering, direct type-casts between my_combined_type and my_signal_type, may not work. Using mixed representations for the same signal is not encouraged, since the potential for errors is high.
- As a matter of form, the type definitions are not required but can be used to enhance readability.
- When the sizes of basic data types are nonstandard, specifications using unambiguous, pseudo-code data-types (i.e. Int32 for a signed 32-bit integer) should be used.
- Designers must be aware of any byte-ordering concerns in the system. It is standard practice for GRRDE modules to convert all output data to so-called network-ordered (big-endian) format. Any deviations from this practice should be noted.
- The units used in any physical quantities should also be specified in this section. They can be included as source-code comments:

```
typedef struct {
double dPosition[3]; //Geocentric, Cartesian (in m)
} my_struct
```

Period

This quantity is only meaningful for time-triggered elements. For an output specification, the period defines the time interval in ms between successive updates of a data-product. The period specification for input elements describes the period of the message contract that will be delivered. In general the relation

$$P_{\text{source}} \leq P_{\text{sink}} \tag{4.1}$$

should be maintained. GRRDE permits the source period to be longer than the sink period, but this results in the delivery of redundant, repeated data.

ARGC, ARGV

The GSCA permits a limited degree of parameterization during contract setup. The internal mechanisms allow one numeric parameter, and one arbitrary parameter to be specified during contract setup. These parameters are passed to the dispatch function when the contract is triggered. ARGC is a 32-bit integer, and ARGV is a character string (although it can carry arbitrary contents). The interface specification for an output must specify whether the parameters are used or ignored, the range of permitted values, and how they are interpreted. Input specifications must explain the values that will be used when the contract is established. It is not necessary to specify the precise values during the interface definition process, but the origins of the quantities should be clear (i.e. UID, etc.).

Interface Definition Filename

To simplify interfacing between modules, it is helpful to share the data structure declarations and signal-number definitions. GRRDE has adopted the OSE practice of naming signal definition files with a "`.sig`" extension. These files are included in a program through the customary "`#include`" directive. Unless specified otherwise, all definitions for a given SN will be found in a common file.

Example specifications are provided in Appendix B.

4.5 Summary

This chapter began by introducing the GRRDE approach to analysis and design. This was followed by an examination of the suggested implementation approach and tools. The emphasis on independent, well specified, functional modules has been stressed throughout the process. Particular attention has been paid to how aspects of design reflect in latter stages. Although module independence is critical to producing better code, it is also important that the design not become fragmented. Chapter 7 revisits the process of analysis and design with an eye for engineering the architecture of the simulation as a whole. Meanwhile, we wish to examine the performance of the middleware system.

The GRRDE publish subscribe system provides simulations with low-overhead, low-maintenance state mobility. The next few chapters explore these properties of the GRRDE system. We begin with formal specification and validation of the essential publish-subscribe algorithms in Chapter 5. This is followed by testing results in Chapter 6. By assembling the program modules and communication primitives, a complete simulation design can be synthesized. Special attention must be paid to the flight-software/simulator interface to ensure that the deployment process is as straightforward as possible. These system-level issues are discussed in Chapter 7.

Chapter 5

FORMAL VALIDATION OF GRRDE RUN-TIME SERVICES

The defining characteristic that sets embedded software apart from conventional programs is the focus on determinism (see Section 2.1.2). As part of the validation process, software systems are scrutinized using a variety of methods. Not only must the program's behaviour be logically correct, but its usage of system resources, such as CPU time or memory, must be well characterized and bounded. Therefore, tools like GRRDE, that are intended to support embedded software development, must lend themselves to the same sort of analysis and rigor.

This chapter describes the formal validation of the GRRDE runtime services. Together with run-time testing presented in Chapter 6, these derivations provide precise characterization of GRRDE to support subsequent analysis in the users' applications. We begin by considering a formal analysis of the GRRDE algorithms. In Chapter 7, we consider how these results can be used in a wider real-time analysis framework.

5.1 Formal Analysis Using General Timed Automata

General Timed Automata (GTA) is an analysis technique suitable for partially-synchronous, distributed systems. In this section we apply GTA modelling technique to the publish-subscribe services provided by GRRDE. This serves two objectives. First, by constructing an *abstract automaton* describing the GRRDE services, we have and unambiguous, formal specification of the properties that the system guarantees. Second, we

produce hierarchical proofs that validate the *correctness* of the algorithms implemented in the GRRDE source code. We have proved basic timing and safety properties of both time-triggered and change-triggered contracts. Timing properties are parameterized by constants that are measured in the next chapter (Chapter 6).

5.1.1 Context of GTA Modelling

Hierarchical proofs can be used as a means of increasing the utility of automaton-based formal methods. Readers unfamiliar with the automaton-based proof techniques are encouraged to read Lynch [Lynch, 1996]. This proof technique requires several steps. First, we specify an abstract, or *specification*, automaton. This automaton must satisfy the properties that we are interested in, yet make as little commitment as possible to the way in which the properties are satisfied. These properties are typically stated as logical invariants and proven through induction. Next, we formulate the detailed, or *implementation*, automaton. This may be a single automaton or a composition. Showing a rigorous correspondence, or *simulation relation,* between the states and behaviours of the specification and implementation automata, is enough to prove that the implementation automaton satisfies the same invariants and properties. The premise behind this approach is that it is usually more tractable to prove properties of the specification than the implementation itself.

This process can be repeated several times through successive steps of simulation and composition (Figure 5.1). If a precise simulation relation can be provided for each stage, the underlying constructions can be formally verified. Thus, you could prove that your algorithm matches your specification, that your source code correctly implements an algorithm, and that your compiler produces exactly the correct machine instructions from code that you write [Erkkinen, 1999]. In theory, this process can be extended all the way from specification to hardware, but the effort required to do this for realistic systems is prohibitive.

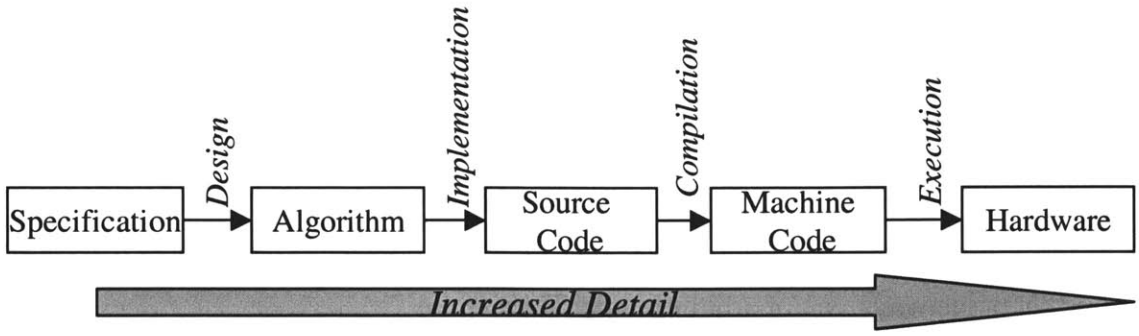


Figure 5.1 Levels of Abstraction in simulation proofs. If the semantics at each stage can be specified precisely, each level can be formally verified.

Several factors contribute to these limitations: state explosion makes large models unwieldy and the lack of formal semantics for some computer languages hampers the usefulness of the process. More importantly, the more complex the system, the easier it is to make mistakes in the formal analysis [Tanenbaum, 1976]. Finally, since most software-related failures can be traced to design and implementation errors [Lutz, 1992], conventional testing is less costly and more effective than exhaustive formal analysis during these development stages.

This leads us to consider how to effectively use formal analysis to support verification of the GRRDE services. How much analysis do we need? We have used GTA models to prove properties of the GRRDE service specifications and algorithms. These stages are easiest to model and are most crucial to the elimination of system level errors. Thus, we gain the maximum benefit from a modest amount of effort.

In Section 5.2, we present our analysis of the GRRDE specifications. We examine the publish-subscribe services as abstractions and establish key timing and correctness properties. We discuss both time-triggered and change-triggered contracts. In Section 5.3, we present detailed automata that represent the implementation of GRRDE and show that they maintain the same properties.

5.1.2 Notational Conventions

Before embarking on a detailed discussion the automata models, it is worth clarifying a few mathematical conventions that we have employed in our proofs.

Sequence Data Types

Many of the models use the primitive data type known as a *sequence*. A sequence, S, is an ordered set of elements, s_n, of arbitrary type. Thus:

$$S = \{s_0, s_1, \ldots, s_{N-1}\} = \{s_n\}\Big|_{n=0}^{N-1} \tag{5.1}$$

The following sequence operators are useful. Given the sequence S, the *head* operator refers to the first element:

$$head(S) = s_0 \tag{5.2}$$

while the *tail* operator refers to the remainder of the sequence. I.e.,

$$tail(S) = \{s_n\}\Big|_{n=1}^{N-1} \tag{5.3}$$

Similarly, the *last* operator returns the final element,

$$last(S) = s_{N-1} \tag{5.4}$$

and the *init* operator returns the start of the sequence:

$$init(S) = \{s_n\}\Big|_{n=0}^{N-2} \tag{5.5}$$

We may also append,

$$S \vdash a \equiv \{s_0, s_1, \ldots, s_{N-1}, a\} \tag{5.6}$$

and prepend elements

(5.7)

Two sequences can also be concatenated to form a new sequence:

$$A \parallel B \equiv \{a_0, a_1, \ldots, a_{M-1}, b_0, b_1, \ldots, b_{N-1}\} \tag{5.8}$$

Relations Between Sequences and Elements

Elements that are members of a sequence, $S = \{s_n\}$ can obey a predecessor relation:

$$a \preceq b \Leftrightarrow a = s_i, b = s_j, i \leq j \tag{5.9}$$

Sequences can also be related to one another. We define the following two relations between sequences. It is unclear whether there are canonical notations for these relations, but the following definitions are use in our proofs.

We say that A is a subsequence of B, if the elements of A are contiguous elements of B, i.e.:

$$A = \{a_0, a_1, \ldots, a_{M-1}\} = \{b_n\}\Big|_{n=j}^{j+M-1} \tag{5.10}$$

This is denoted $A \preceq B$.

A looser relation is to say that A is a subset of B (Denoted $A \subseteq B$). This allows us to construct A from B by omitting arbitrary elements. Thus all elements of A are found in B,

$$\forall s, s \in A \Rightarrow s \in B \tag{5.11}$$

and this preserves the pair-wise successor relations between elements:

$$\forall (a_i, a_j) \in A, a_i \preceq a_j \Rightarrow \exists (b_m, b_n) \in B, a_i = b_m \wedge a_j = b_n \wedge b_m \preceq b_n \tag{5.12}$$

5.1.3 IOA Pseudocode

The automata specifications presented in this chapter were developed with the *Input-Output Automata Toolkit (IOA Toolkit) [Garland & Lynch, 2000]*. IOA defines a structured pseudocode in which the user can write automata models. This package performs syntax checking, type-setting and can interface with a number of theorem proving tools.

Although the advanced features of the IOA Toolkit do not yet support GTA models, we adopted this pseudocode standard to give our specifications a consistent appearance.

5.2 Analysis of GRRDE Specifications

GRRDE provides two types (SECTION) of communications services: time-triggered and change-triggered. Modules seeking to establish contracts with server blocks, first looked up the services and then initiated a contract request. Stepping back from the details of implementation that we saw in Chapter 4, we can consider a slightly abstracted version of this service interaction (Figure 5.2, Figure 5.3).

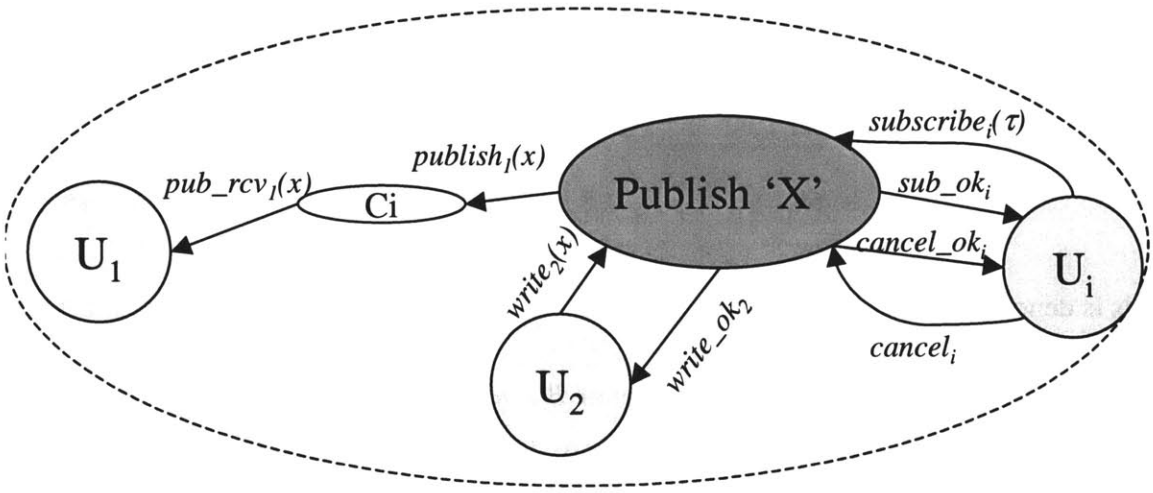


Figure 5.2 Time-Triggered Publish composition. Model consists of user automata (U_i), as well as the service (Publish) and channel (C) automata.

These figures shows three types of automata. The finite set of user processes, $U_i, i \in I$, represent the interface to the publish-subscribe system. We do not model any processing by the user, simply the interaction with the GRRDE service. Rather than depict the subscription selection, the service automaton, $Publish$, simply provides subscriptions to the single variable, X. We can view a full system as a parallel composition of several of these service automata, but a single type of subscription is sufficient for purposes of validation.

The modelling of the delivery mechanism includes a reliable, *first-in-first-out (FIFO)* channel automaton C_i to capture effects of finite propagation time.

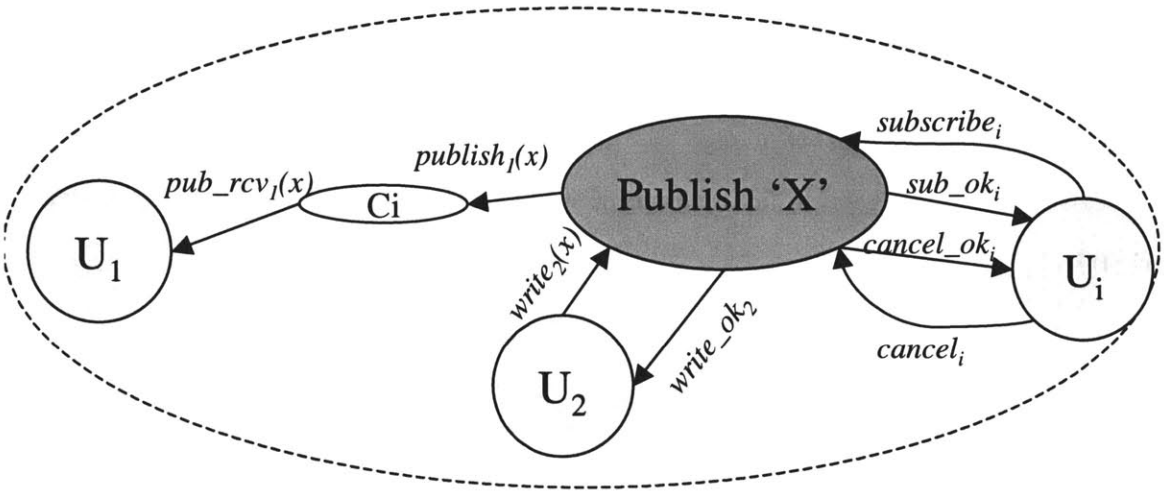


Figure 5.3 The Change-Triggered Automata. Identical to time-triggered service, save that *subscribe* is not parameterized.

Three types of interactions are depicted. The first concerns the subscription process. This includes the *subscribe*, *sub_ok*, *cancel*, and *cancel_ok* actions. The time-triggered subscribe action is parameterized by the quantity, τ, which represents the requested delivery period. The *Cancel* action ends a current subscription and the other two actions are simply confirmations. The second type of interactions allows writing a new value, v, to X, and the associated confirmation. In this model we allow any process to perform a write action even though write permissions are restricted in the actual GRRDE environment. The last phenomenon modelled is the delivery process. Publish notifications are emitted from the publish server and propagate through the channel automaton C_i to the user. It should be noted that all of these actions are available to each user process. The figure just separates them for simplicity.

5.2.1 Channel Modelling

One might notice from the preceding figures that the message propagation channels are explicitly modelled only for publish actions and not for the other interactions. This was done to simplify the analysis. Inter-process traffic under the OSE operating system is essentially multiplexed. Messages from all senders are deposited in a common input queue at the destination process. However, we model most interactions as direct connections between automata. It is worth taking a moment to examine the justification for this idealization.

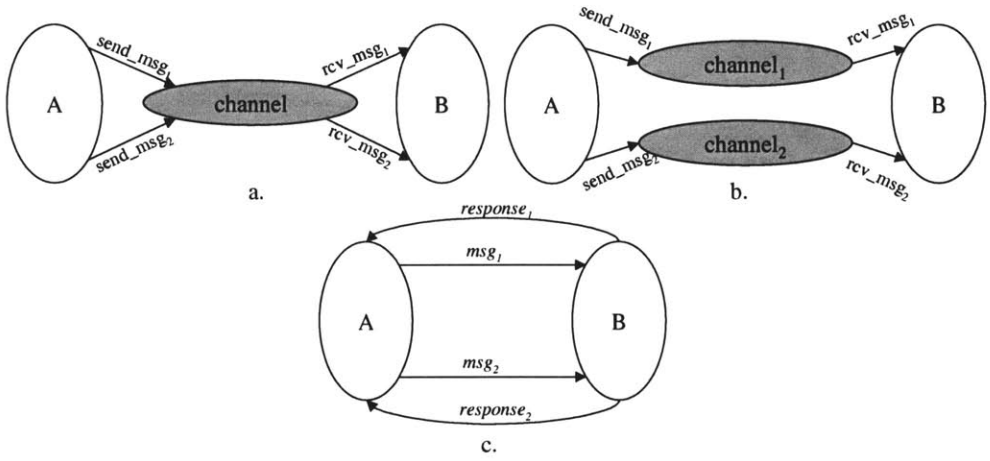


Figure 5.4 Abstractions of message channels. Examples include multiplexed channels, (a), parallel channels (b), and direct I/O connections, (c).

Figure 5.4 shows the progressive abstractions that we use to model inter-process communication under the OSE operating system. The most accurate representation is given by Figure 5.4a. All inter-process messages are multiplexed onto common FIFO channels. We contend that this behaviour can be represented by either parallel FIFO channels (Figure 5.4b) or direct I/O connections (Figure 5.4c). Parallel channels are appropriate when propagation time is important and the effect of the input actions at **B** are independent. I.e. for all states s the execution fragments $\alpha_1 = (s, rcv_1, s_2, rcv_2, s')$ and $\alpha_2 = (s, rcv_2, s_3, rcv_1, s')$ give the same end state s'. In other situations, where ordering

is important, we make the assumption that only one message of each type is in transit at a time and that these actions have paired acknowledgements. In these situations we can convert the channels into direct I/O connections[1].

This reasoning behind this last argument can be expanded. Consider the interaction between two automata (Figure 5.5). In this example there are two sets of grouped interactions: $(invoke_1, response_1)$, and $(invoke_2, response_2)$. The automaton **B** processes incoming invocations independently taking several internal actions. However, both inputs may affect common state variables of **B**. In distributed systems, where asynchrony makes timing unpredictable it is common to assume that interactions are *well-formed.* Well-formedness assumptions restrict the allowable interactions between automata. For instance we may impose that following an $invoke_1$ action, automaton **A** must wait to receive a $response_1$ before generating another $invoke_1$, but is free to make an $invoke_2$ in the meantime. Alternately, we might enforce that any invoke must receive a response before further outputs can take place. We might also make further restrictions and require interactions to follow the pattern of $invoke_1$, $response_1$, $invoke_2$, $response_2$.

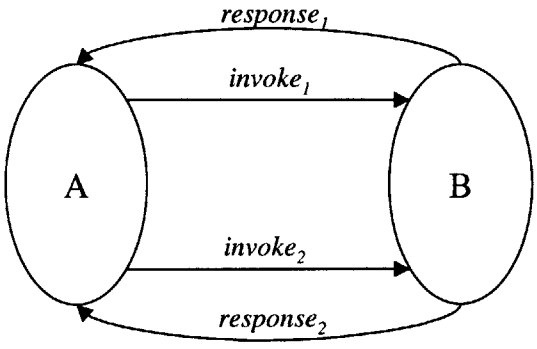


Figure 5.5 Parallel automata interactions.

1. When only one message is in transit at a time, we can account for transmission delays as an extra internal step at either **A** or **B.**

The purpose of these assumptions is twofold. First, they make automata specifications simpler, since we do not need exhaustively enumerate the handling of malformed interactions[1]. Second, enforcing invoke-response pairing is a principle of good algorithm design and allows the automata to maintain a consistent view of system operations. Thus, when **A** waits for a $response_1$, it is assured that the preceding $invoke_1$ has been processed by **B**. Furthermore, both **A** and **B** will have the same view of the global action sequence. Consequently, the GRRDE specifications make well-formedness assumptions about interactions with client processes and can employ direct connections in many situations

This abstraction process greatly simplifies the modelling of GRRDE services. Removing the elaborate OSE message queues allows us to model direct, intuitive interfaces to the service automata. The parts of the user's algorithms that we have abstracted away, are typically just a conditional branching structure based on the identifiers in the incoming messages. Correctness of this code could be established with additional formal modelling or standard testing techniques.

5.2.2 Time-Triggered Specification

A pictorial representation of the time-triggered subscription service was given in Figure 5.2. Note that most of the interaction between users and the service automaton is through direct commands and responses. Only the dispatched messages utilize a channel for delivery. This allows us to reason about dispatch messages in transit. Transit times for commands and responses are accounted for implicitly. Our aim, in analyzing the code, is to prove basic safety and consistency properties of the dispatched values as well as timing properties for the delivery mechanism.

IOA pseudocode for the specification automaton is given in Figure 5.8. This automaton encapsulates and publishes the shared variable $x \in X$. This variable is a tuple $X \equiv [v, seqno], v \in V, seqno \in N$ where v is the actual data component of the service

1. The designer may want to include such logic when considering tolerance for lost messages, failures, etc., but it is useful to consider failure-free situations first.

and is of arbitrary type, while *seqno* is a unique sequence number. Each action, $write(i, v')$, causes a change in stored value of x such that $x'.v = v'$ and $x'.seqno = x.seqno + 1$. Sequence numbers allow us to precisely relate the contents of publish notifications to a unique write event.

This automaton is actually a manual composition of the three types of automata depicted earlier. This composition includes the actual service automaton *publish*, as well as the channel automata C_i, and the user automata, U_i. Since input actions are subsumed by composition, all of actions are considered outputs[1]. The state declarations describe the internal state of the composed automaton. We compose the automaton in this way to permit analysis of the traces of the central *publish* automaton. In this analysis the user automata exist only to maintain an external log of subscription notifications and to enforce well-formedness on system invocations.

There are several types of state variables in the pseudocode:

- Constants. The IOA language used in these examples does not allow the definition of symbolic constants (e.g. the use of Δ_{max} to represent a upper bound on delay). These variable are used mostly to specify parameters in invariant statements. The numeric values in the pseudocode were used only for testing purposes and should be ignored.
- Time Bounds. GTA automata use explicit variables such as *first* or *last* to restrict when certain actions can take place. These variables are a standard feature of GTA models, but do not represent "real" variables.
- Log Variables. Many interesting automaton properties are naturally described by discussing the system traces and execution (i.e. the steps an algorithm takes). However, state invariants are the most straightforward properties to prove. We can convert trace properties into state invariants by introducing artificial log variables. Thus, as an automaton takes an action, it records an entry in this state variable. This useful contrivance creates a common framework for all of our proofs, but these variables are not reflected in our software.

1. Readers unfamiliar with automata techniques are encouraged to refer to Lynch, 1996 for a summary of key principles and techniques

- Constraints. Well-formedness assumptions about the automaton interactions can be made explicit in the specification. Certain state variables are attributed to the user processes and inhibit further commands while confirmation for a current command is still pending.
- Core Variables. These state variables are intended to have some physical realization in an implemented system. Abstract, specification automata will typically have few state components of this type.

We also define the following additional variable types:

- $WriteRec \equiv [x, i, t], x \in X, i \in I, t \in \Re^{\geq 0}$. This tuple is used to keep records of the incoming *write* actions. We record the value, x, the process index, i, and the time t.
- $TimeBound \equiv [lb, ub], lb, ub \in \Re^{\geq 0}$. This tuple is used to record lower and upper timing bounds. These variables can record relative or absolute time.
- $TxRec \equiv [x, bound], x \in X, bound \in TimeBound$. These tuples are used to transmit published values through the transit queues from the service automaton to the clients. The first element contains the published data while the second gives the earliest and latest delivery times.
- $RcvRec \equiv [x, k, t], x \in X, k \in Z^{+}, t \in \Re^{\geq 0}$. These variables are used to log message delivery in the user automata. We record the data, x, a message count, k, and the arrival time, t.

Figure 5.9 summarizes the state invariants of the publish automaton. The first set of proofs establish temporal consistency of the values delivered to the clients. We require that the unique values delivered to the clients represent an ordered subset of the values written to the *publish* automaton This property is depicted graphically in Figure 5.6. Thus, client records may skip or duplicate values written to the central automaton, but the sequence of values may not be reordered. The second set of proofs provides bounds on the temporal accuracy of the client values. This bound defines a sampling window. Sometime within this window, the true value of x must have been equal to the observed client value. The last set of proofs describe the periodic performance of the subscription service.

Lemma 5.2.1: *In every reachable state of publish, all written values of x, save the initial value can be associated with a writing process, i.*

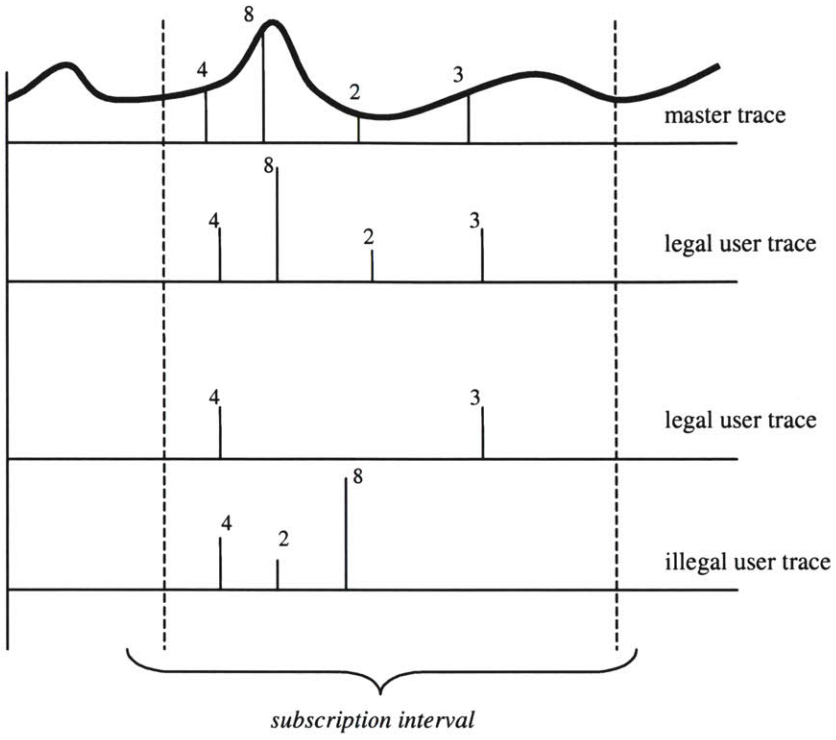


Figure 5.6 Consistency of user traces. The master trace represents the values written to the shared variable x. Legal user traces must be consistent with the ordering of the master trace.

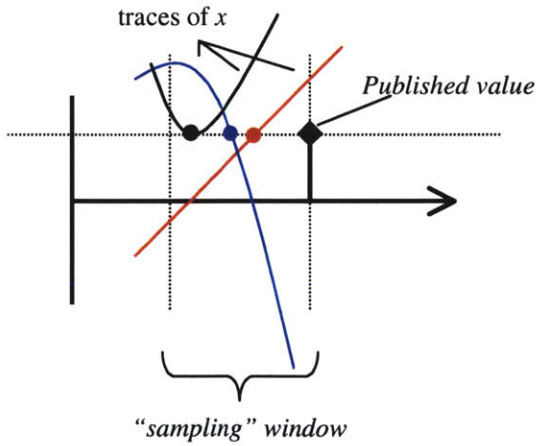


Figure 5.7 Temporal accuracy of client values. At some point within the sampling window, the central value of x must have been equal to the value delivered to the client.

```
uses NonDet
uses Publish_Types
uses Nulled(I)
uses Conversions

automaton Publish(v0 : Int, Inf:Real)
    signature
        output
            subscribe (i: I, tau: Real),
            publish(i:I, T: TxRec),
            pub_rcv(i:I, T: TxRec),
            cancel(i:I),
            sub_ok(i:I),
            cancel_ok(i:I),
            write(i:I, v:V),
            write_ok(i:I)
        internal
            WriteCommit(i:I),
        time-passage
            ν(t: Real) %Time Passage
    states
        %These are really just "contstants"
        sub_ok : TimeBound,
        cancel_ok : TimeBound,
        write_ok : TimeBound,
        xmit: TimeBound,
        pub_jitter : Real,
        WriteCommitTime: TimeBound,
        %Records each client node
        Dest_Seq: Array[I,Seq[RcvRec]] := constant({}),
        PubCount: Array[I, Int] := constant(0),
        StartTime: Array[I, Real] := constant(0),
        %Transit Queues
        Dest_Queue: Array[I, Seq[TxRec]] := constant({}),
        %Housekeeping to ensure well formedness
        Subscribed: Array[I, Bool] := constant(false),
        PendingCommand: Array[I,Command] := constant(nil),
        Period: Array[I,Real] := constant(0),
        PubStarted: Array[I, Bool] := constant(false),
        %Records at server
        WriteLog: Seq[WriteRec] := {} ⊢ [[v0, 0], null, 0],
        %Published Variable
        pub_value : DataRec := [v0,0],
        PendingWriteValue := Array[I,v],
        %Time, and GTA bounds
        now : Real := 0,
        first_cancel_ok: Array[I, Real] := constant(0),
        last_cancel_ok: Array[I, Real] := constant(Inf), %Inf
        first_publish: Array[I, Real] := constant(0),
        last_publish: Array[I, Real] := constant(Inf), % Inf
```

Figure 5.8 Time-Triggered Publish Automaton.

```
    first_sub_ok: Array[I, Real] := constant(0),
    last_sub_ok: Array[I, Real] := constant(Inf), %Inf
    first_write_ok: Array[I, Real] :=constant(0),
    last_write_ok: Array[I, Real] := constant(Inf) %Inf
transitions
    output subscribe(i:I, τ:Real)
        pre PendingCommand[i] = nil ∧ ¬Subscribed[i]
        eff PendingCommand[i] := Sub;
        Subscribed[i] := true;
        Period[i] := τ;
        StartTime[i] := now;
        %Housekeeping
        first_sub_ok[i] := now + sub_ok.lb;
        last_sub_ok[i] := now + sub_ok.ub;
        first_publish[i] := now + Period[i];
        last_publish[i] := first_publish[i] + pub_jitter
    output sub_ok(i:I)
        pre PendingCommand[i] = Sub ∧ now ≥ first_sub_ok[i]
        eff PendingCommand[i] := nil;
        first_sub_ok[i] := 0;
        last_sub_ok[i] := Inf
    output publish(i:I, T:TxRec)
        pre Subscribed[i] ∧ now ≥ first_publish[i]
                          ∧ T = [pub_value,[now + xmit.lb, now + xmit.ub]]
        eff Dest_Queue[i] := Dest_Queue[i] ⊢ T;
        %Housekeeping
        first_publish[i] := first_publish[i] + Period[i];
        last_publish[i] := first_publish[i] + pub_jitter
    output pub_rcv(i:I, T:TxRec)
        pre T = head(Dest_Queue[i])
            ∧ now ≥ head(Dest_Queue[i]).Delivery.lb
            ∧ PendingCommand[i] ≠ Sub
        eff Dest_Queue[i]:= tail(Dest_Queue[i]);
        PubCount[i] := PubCount[i] + 1;
        Dest_Seq[i] := Dest_Seq[i] ⊢ [T.X, PubCount[i], now]
    output cancel(i:I)
        pre Subscribed[i] ∧ PendingCommand[i] = nil
        eff Subscribed[i] := false;
        first_publish[i] := 0;
        last_publish[i] := Inf;
        PendingCommand[i] := Cancel;
        first_cancel_ok[i] := now + cancel_ok.lb;
        last_cancel_ok[i] := now + cancel_ok.ub
    output cancel_ok(i:I)
        pre PendingCommand[i] = Cancel ∧ now ≥ first_cancel_ok[i]
                                       ∧ Dest_Queue[i] = {}
        eff PendingCommand[i] := nil;
        PubCount[i] := 0;
        Dest_Seq[i] := {};
        first_cancel_ok[i] := 0;
```

Figure 5.8 Time-Triggered Publish Automaton.

```
    last_cancel_ok[i] := Inf
output write(i:I, v:V)
    pre PendingCommand[i] = nil
    eff PendingWriteValue[i] := Data
    first_write_commit := now + WriteCommmitTime.lb;
    last_write_commit := now + WriteCommitTime.ub;
    PendingCommand[i] := Commit
internal WriteCommit(i:I)
    pre PendingCommand[i] = Commit
    eff pub_value := [Data, pub_value.SeqNo + 1];
    WriteLog := WriteLog ⊢ [pub_value, i, now];
    PendingCommand[i] := Write;
    first_write_ok[i] := now + write_ok.lb;
    last_write_ok[i] := now + write_ok.ub
output write_ok(i:I)
    pre PendingCommand[i] = Write ∧ now ≥ first_write_ok[i]
    eff PendingCommand[i] := nil;
    first_write_ok[i] := 0;
    last_write_ok[i] := Inf
time-passage ν(t: Real)
    pre (∀ i:I ((now + t) < last_write_ok[i]
        ∧ (now + t) < last_cancel_ok[i]
        ∧ (now+ t) < last_sub_ok[i]
        ∧ (now + t) < last_publish[i]
        ∧ (len(Dest_Queue[i]) > 0 ⇒
          ((now + t) < head(Dest_Queue[i]).Delivery.ub))))
    eff now := now + t
```

Figure 5.8 Time-Triggered Publish Automaton.

Proof: This is a fairly trivial assertion and is formally stated as Invariant I1. We consider an induction on the states of *Publish*. We can see from the automaton specification that the invariant holds in the start state. Assuming that the invariant holds in state s we consider the transition (s, π, s') and prove that the invariant hold in the final state s'. The property is vacuously true in all actions, π, save *write*, since no other actions alter the *WriteLog* variable. Considering this case in more detail we have:

- Case $\pi = write(i, v)$. The invariant is clearly true in the final state since the appended element of *WriteLog* is tagged with the process index i.

Therefore the invariant assertion holds for all reachable states of *Publish*.

Lemma 5.2.2: *In every reachable state of automaton publish, all values, x' in transit to clients must have been previously written (i.e. appear in the WriteLog).*

```
invariant I1 of Publish:
  ∀ W: WriteRec (W ∈ WriteLog ⇒ (W.i = null ⇔ W = head(WriteLog)))

invariant I2 of Publish:
  ∀ T: TxRec (∀ i:I ((T ∈ Dest_Queue[i]) ⇒
              (∃ W:WriteRec (W ∈ WriteLog ∧ W.X = R.X))))

invariant I3 of Publish:
  ∀ R: RcvRec (∀ i:I ((R ∈ Dest_Seq[i]) ⇒
               (∃ W:WriteRec (W ∈ WriteLog ∧ W.X = R.X))))

invariant I4 of Publish:
  ∀ i:I (∀ m: Int (∀ n: Int ((m ≥ 0 ∧ m < len(Dest_Queue[i])
                              ∧ n ≥ 0 ∧ n < len(Dest_Queue[i]) ∧ m < n) ⇒
                             Dest_Queue[i][m].X.SeqNo ≤ Dest_Queue[i][n].X.SeqNo)))

invariant I5 of Publish:
  ∀ i:I (∀ m: Int (∀ n: Int ((m ≥ 0 ∧ m < len(Dest_Seq[i])
                              ∧ n ≥ 0 ∧ n < len(Dest_Seq[i])
                              ∧ m < n) ⇒ Dest_Seq[i][m].X.SeqNo ≤ Dest_Seq[i][n].X.SeqNo)))

invariant I6 of Publish:
  ∀ t1:Real (∀ i:I (∀ R:RcvRec ((t1 = now ∧ R.t = t1 ∧ R ∈ Dest_Seq[i]) ⇒
             (pub_value = R.X ∨
             (head(WriteLog).X ≠ R.X ∧ head(WriteLog).t ≥ now - xmit.ub)))))

invariant I7 of Publish:
  ∀ i:I (now ≥ StartTime[i])

invariant I8 of Publish:
  ∀ i:I (PubCount[i] = len(Dest_Seq[i]))

invariant I9 of Publish:
  ∀ i:I ((len(Dest_Queue[i]) ≠ 0) ⇒ subscribed[i])

invariant I10 of Publish:
  ∀ i:I (∃ k:Int ((Subscribed[i] ∧ (k=floor((now-StartTime[i]-pub_jitter)/Period[i]))) ⇒
         (k≥PubCount[i] ∨ (k = (PubCount[i] + 1) ∧ (∃ T:TxRec (T ∈ Dest_Queue[i]
          ∧ T.Delivery.ub ≤ (StartTime[i] + k*Period[i] + pub_jitter + xmit.ub)))))))

invariant I11 of Publish:
  ∀ i:I (∀ R:RcvRec ((R ∈ Dest_Seq[i]) ⇒
                (R.t ≥ (StartTime[i] + R.K*Period[i] + xmit.lb)
                ∧ R.t ≤ (StartTime[i] + R.K*Period[i] + pub_jitter + xmit.ub))))
```

Figure 5.9 Time-Triggered Automaton invariants.

Proof: This is invariant I2. Consider induction on the states of *publish*. Since the message channels ($DestQueue_i$) are initially empty, the assertion is clearly true in the initial state. Considering the inductive step we examine the transition (s, π, s'). The assertion is vacuously true for all transitions save $publish_i(T)$ and $pub\_rcv_i(T)$. We examine these transitions separately.

- Case $\pi = publish_i(T)$. Since the transition specifies that $T.x = head(WriteLog).x$, the final state s' must satisfy the invariant.

- Case $\pi = pub\_rcv_i(T)$. This action only removes elements from $DestQueue_i$, thus s' must satisfy I2.

Lemma 5.2.3: *In every reachable state of publish, all published values received by user automata must have been previously written (i.e. appear in the WriteLog).*

Proof: We consider induction on the states of *publish*. This invariant is I3. In the initial state of *publish*, the client records are empty so the assertion is clearly true. For the inductive step in the proof we consider the state transition (s, π, s'). The only actions that modify the client record $DestSeq_i$ are $cancel\_ok(i)$ and $pub\_rcv(i, T)$. For the other actions, the invariant assertion is vacuously true.

- Case $\pi = cancel\_ok_i$. Since in s', the client log is empty, i.e. $\neg\exists R, R \in DestSeq_i$, and the invariant I3 still holds.
- Case $\pi = pub\_rcv_i(T)$. We know from Lemma 5.2.2. that all entries in the channel queues must correspond to *WriteLog* entries. Therefore when these messages are received by the user processes, the correspondence must be maintained.

Lemma 5.2.4: *In every reachable state of publish, the sequence numbers of messages in transit are non-decreasing.*

Proof: We prove invariant I4 by induction on the states of *publish*. The predicate to the implication simply picks two valid, ordered indices of messages in transit. Clearly the invariant holds in the initial state, since the messages channels are initially empty. Assuming that the assertion holds in state s we consider the transition (s, π, s'). Since all actions save for $pub\_rcv_i(T)$ and $publish_i(T)$, leave the variables $DestQueue_i$ unchanged, they vacuously satisfy the invariant. We consider the two remaining cases.

- Case $\pi = publish_i(T)$. We need only consider the relation between T and the entry $tail(s.DestQueue_i)$, since by the inductive hypothesis, the rest of the sequence has non-decreasing sequence numbers. If $tail(s.DestQueue_i).X = T.X$, the invariant is clearly true. Conversely, if $tail(s.DestQueue_i).X \neq T.X$ then the invariant must also be true since $T.X = tail(WriteLog).X$ and the *WriteLog* sequence numbers are monotonically increasing.

- Case $\pi = pub\_rcv_i(T)$. This action only removes elements from $DestQueue_i$, and thus s' must satisfy I4.

Lemma 5.2.5: *In every reachable state of publish, the sequence numbers of messages delivered to user processes are non-decreasing.*

Proof: In order to prove invariant I5, we consider induction on the states of *publish*. The predicate to the invariant implication simply picks two valid, ordered indices of messages in the destination log. Initially, the invariant holds since the $DestSeq_i$ sequences are empty. We consider the state transition (s, π, s') for the inductive step. All transitions π, save $cancel\_ok(i)$ and $pub\_rcv_i(T)$ leave the variables $DestSeq_i$ unchanged. We examine the invariant assertion for the remaining cases.

- Case $\pi = cancel\_ok_i$. Since in s', the client log is empty, i.e. $\neg\exists R, R \in DestSeq_i$, the invariant I4 holds.
- Case $\pi = pub\_rcv_i(T)$. To prove that the invariant holds through this transition, we consider three sub-cases. First, we consider the case where $s.DestSeq_i = \emptyset$, i.e. the destination record is initially empty. This case clearly satisfies the assertion since the resulting destination sequence has only a single element, i.e. $s'.DestSeq_i = \{R\}, R \in RcvRec, R.x = T.x$. In the second sub-case, we receive a duplicate message, i.e. $T.x = head(DestSeq_i).x$. This trivially satisfies the assertion. Lastly, we must examine the case where we receive a new value. Since from Lemma 5.2.4, we know that the sequence numbers in $DestQueue_i$ are non-decreasing, this property is preserved as we record the incoming values.

Theorem 5.2.1: *In every reachable state of publish, the sequence of unique values recorded by subscribed users, constitute an ordered subset previous values of the shared variable x. I.e.* $\forall i \in I, Unique(DestSeq_i.X) \subseteq \{WriteLog.X\}$.

Proof: This property follows directly from Lemma 5.2.3, which shows the correspondence between elements of the user and master records, and from Lemma 5.2.5, which ensures that the ordering of the elements is preserved. □

We now establish the temporal correspondence between the values recorded at the user automata and the master log.

Lemma 5.2.6: *In every reachable state of publish, if a published value, x_1, is received at time, t_1, by a user process, then either $x_1 = PubValue$, or some process has written a new value to X with the last xmit.ub time units.*

Proof: At some time during the time interval defined by the lemma, the true published value must have agreed with the received value. We will prove invariant I6 through induction on the states of *publish*. This invariant is concerned with the state of the system at the instant a value is received by the user. In the initial state of the automaton, this assertion holds. We are then left to investigate the inductive step. We assume that the assertion holds in state s and consider the transition (s, π, s'). There are four non-trivial cases to consider:

- Case $\pi = pub\_rcv_i(T)$. Two possibilities exist. If $T.X = R.X$, we have simply duplicated an entry in the user log and the property holds from the invariant assertion. Alternately, $T.X \neq R.X$. Since we know from Lemma 5.2.1 that we can associate a *write* action with T, and each message spends at most *xmit.ub* time in transit, therefore the assertion must also hold in s'.
- Case $\pi = cancel\_ok_i$. Since $s'.DestSeq_i = \varnothing$, the assertion is trivially satisfied.
- Case $\pi = WriteCommit_i$. This action clearly satisfies the assertion, since it directly adds an element to *WriteLog*.
- Case $\pi = \nu(t)$. Since $t > 0$, $s'.now \neq t_1$. This causes the predicate clause of the assertion implication to be false. Consequently the assertion becomes trivially true.

Theorem 5.2.2: *In every reachable state of publish, the each value delivered to the user automata reflects a true value of the publish variable within the preceding time window of width xmit.ub.*

Proof: This property follows directly from the assertion described in Lemma 5.2.6. □

These two theorems establish basic properties of the GRRDE services. Both types of subscriptions possess the properties described above. Time-triggered contracts possess addi-

tional behaviours that characterize their periodic nature. We develop these properties with a further set of state invariants.

Lemma 5.2.7: *In every reachable state of $publish$, the current time is at least as great as the subscription-start time of any subscribed automaton.*

Proof: Assertion I7 is a fairly trivial property. The induction proof proceeds without complexity. The invariant holds in the start state. Thereafter, the only changes to $start\_time_i$ is to set it equal to the current time

Lemma 5.2.8: *In every reachable state of $publish$, the variable $pubcount_i$ accurately reflects the number of elements in the $DestSeq_i$ log.*

Proof: Like the preceding assertion, I8 is proven by a trivial induction on the states of $publish$ .This assertion is clearly true in the start state. The only actions that modify either variable are $pub\_rcv_i(T)$ and $cancel\_ok_i$. We observe from the specification that the relevant variables are modified together.

Lemma 5.2.9: *In every reachable state of $publish$, only subscribed automata have non-empty message channels.*

Proof: Invariant I9 follows from induction on the states of $publish$. In the start state, no processes are subscribed and all channels are empty. Thus, the assertion is satisfied. For the inductive step, we assume that the assertion holds for state s, and consider the transition (s, π, s'). For many actions the assertion is vacuously true. The remaining actions must be considered individually:

- Case $\pi = cancel\_ok_i$. The precondition for this action requires that message channel be empty. Thus, despite the fact that $s'.subscribed_i = false$, the assertion holds[1].

1. This behaviour is actually a bit of a contrivance. The precondition that inhibits the $cancel\_ok_i$ action until the message queue is empty is actual an enforcement of the well-formedness assumptions.

- Case $\pi = pub\_rcv_i(T)$. Since this action removes elements from the user channels, it maintains the invariant. If T is the only element of $DestQueue_i$, I9 is trivially satisfied. Otherwise, we can observe that through the transition $s.subscribed_i = s'.subscribed_i = true$. This also satisfies the invariant.
- Case $\pi = publish_i(T)$. This action satisfies the invariant since the precondition for the action includes the provision that $subscribed_i = true$

Lemma 5.2.10: *In every reachable state of publish, for all subscribed processes, the k_i-th publish event must occur between $StartTime_i \cdot k_i$ and $StartTime_i \cdot k_i + pub\_jitter$.*

Proof: To prove invariant I10, we consider induction on the states of *publish*. The predicate to the invariant predicate selects an appropriate value of k. This value is compared to the count of received messages and current time bounds. Initially the invariant is satisfied since there are no subscribed processes. For the inductive portion of the proof we consider the transition (s, π, s'). The non-trivial actions are considered below.

- Case $\pi = publish_i(T)$. From the inductive hypothesis, we must have already witnessed $k_i - 1$ publish events. If this action is enabled then, the lower bound of the invariant holds. From the inductive hypothesis the upper bound must also hold as well. Therefore the assertion holds in s'.
- Case $\pi = \nu(t)$. If this action is enabled, then from the preconditions we know that $s.now + t < LastPublish_i$. Therefore, this action is not enabled if the time-step would carry *now* beyond the deadline for the k_i-th publish event. Thus, the assertion is valid.
- Case $\pi = cancel\_ok_i$. Since this action un-subscribes a user, the invariant is satisfied in s'.

Lemma 5.2.11: *In every reachable state of publish, the k_i-th publish message delivered to a subscribed user, is received between $now = StartTime_i + k_i \cdot Period_i + xmit.lb$ and $now = StartTime_i + k_i \cdot Period_i + xmit.ub + pub\_jitter$.*

Proof: To prove invariant I11, we consider an induction on the states of *publish*. Initially the assertion is true since there are no subscribers. For the inductive step, we assume that the assertion holds in state s and consider the transition (s, π, s'). We examine the following actions:

- Case $\pi = pub\_rcv_i(T)$. To satisfy the lower timing bound we note that from Lemma 5.2.10, that the earliest corresponding *Publish* event was at $k_i \cdot Period_i$ and the transit time is at least $xmit.lb$. The upper bound follows from a similar argument.
- Case $\pi = cancel\_ok_i$. This unsubscribes process i, thus satisfying the assertion

Theorem 5.2.3: *Subscribers to time-triggered contracts receive periodic publish messages with bounded jitter* $(pub\_jitter + (xmit.ub - xmit.lb)$

Proof: This property follows directly from the periodic trace history (Lemma 5.2.11), the timely publishing of each value (Lemma 5.2.10), and the bounded transmission time.□

5.2.3 Change-Triggered Subscriptions

The IOA code for this automata is given in Figure 5.10 and the invariants are listed in Figure 5.11. Several of the properties derived for the time-triggered automata hold as well for the change-triggered services. These could re-derived for this automaton, but in the interest of brevity we shell omit this duplication. Theorem 5.2.1 and Theorem 5.2.2 remain unchanged. As before, the *publish_on_change* automaton represents the manual composition of the actual master *publish*, composed with channels C_i and users, U_i. We are interested now in developing the property that is distinct to this automaton, namely its "exactly once" behavior. That is, each subscribed process will get exactly one copy of each value written to x.

Lemma 5.2.1: *In every reachable state of publish_on_change, for each write action, and for each subscribed user, there will be a dispatched publish message. This message must be either: waiting to be dispatched, in transit, or received at the user.*

Proof: To prove assertion I9, we consider induction on the states of *publish_on_change*. Initially the invariant is true since there are no subscribed processes. For the inductive step of our proof, we assume that the assertion holds in state s, and consider the transition (s, π, s'). Non-trivial actions are considered below.

```
%This is the abstract specification for the change triggered subscriptions.
uses Conversions
uses Publish_Types
uses Nulled(I)

automaton publish_on_change(v0:V, Inf: Real)
    signature
        output
            subscribe(i:I),
            sub_ok(i:I),
            cancel(i:I),
            cancel_ok(i:I),
            write(i:I, v:V),
            write_ok(i:I),
            publish(i:I, T:TxRec),
            pub_rcv(i:I, T:TxRec)
        internal
            WriteCommit(i:I),
            GenerateMessages(i:I)
        time-passage
            ν(t:Real)
    states
        %Constants
        sub_ok : TimeBound,
        cancel_ok : TimeBound,
        write_ok : TimeBound,
        xmit: TimeBound,
        pub_jitter :Real,
        dispatch_bound :,
        %Records each client node
        Dest_Seq: Array[I,Seq[RcvRec]] := constant({}),
        PubCount: Array[I, Int] := constant(0),
        StartTime: Array[I, Real] := constant(0),
        Dest_Queue: Array[I, Seq[TxRec]] := constant({}),
        Dispatch_Seq: Array[I, Seq[TxRec]] := constant({}),
        %Housekeeping to ensure well formedness
        Subscribed: Array[I, Bool] := constant(false),
        PendingCommand: Array[I,Command] := constant(nil),
        Generating:I = NULL,
        %Records at server
        WriteLog: Seq[WriteRec] := {} ⊢ [[v0, 0], null, 0],
        %Published Variable
        pub_value : DataRec := [v0,0],
        PendingWriteValue : Array[I,V] := constant(v0),
        %Time, and GTA bounds
        now : Real := 0,
        first_cancel_ok: Array[I, Real] := constant(0),
        last_cancel_ok: Array[I, Real] := constant(Inf),
        first_sub_ok: Array[I, Real] := constant(0),
        last_sub_ok: Array[I, Real] := constant(Inf),
```

Figure 5.10 Specification for Change-Trigger automaton.

```
    first_write_ok: Array[I, Real] :=constant(0),
    last_write_ok: Array[I, Real] := constant(Inf)
    first_write_commit: Array[I,Real] := constant(0),
    last_write_commit: Array[I,Real] := constant(Inf)
transitions
    output subscribe(i:I)
        pre PendingCommand[i] = nil ∧ ¬Subscribed[i]
        eff PendingCommand[i] := Sub;
        Subscribed[i] := true;
        StartTime[i] := now;
        %Housekeeping
        first_sub_ok[i] := now + sub_ok.lb;
        last_sub_ok[i] := now + sub_ok.ub
    output sub_ok(i:I)
        pre PendingCommand[i] = Sub ∧ now ≥ first_sub_ok[i]
        eff PendingCommand[i] := nil;
        first_sub_ok[i] := 0;
        last_sub_ok[i] := Inf
    output publish(i:I, T:TxRec)
        pre len(Dispatch_Seq[i]) ≠ 0
                        ∧ now ≥ head(Dispatch_Seq[i]).Delivery.lb
                        ∧ T = [head(Dispatch_Seq[i]).X,[now + xmit.lb, now + xmit.ub]]
        eff Dest_Queue[i] := Dest_Queue[i] ⊢ T;
        Dispatch_Seq[i] := tail(Dispatch_Seq[i])
    output pub_rcv(i:I, T:TxRec)
        pre Rec = head(Dest_Queue[i])
            ∧ now ≥ head(Dest_Queue[i]).Delivery.lb
            ∧ PendingCommand[i] ≠ Sub
        eff Dest_Queue[i]:= tail(Dest_Queue[i]);
        PubCount[i] := PubCount[i] + 1;
        Dest_Seq[i] := Dest_Seq[i] ⊢ [T.X, PubCount[i], now]
    output cancel(i:I)
        pre Subscribed[i] ∧ PendingCommand[i] = nil
        eff Subscribed[i] := false;
        PendingCommand[i] := Cancel;
        first_cancel_ok[i] := now + cancel_ok.lb;
        last_cancel_ok[i] := now + cancel_ok.ub
    output cancel_ok(i:I)
        pre PendingCommand[i] = Cancel ∧ now ≥ first_cancel_ok[i]
                                    ∧ Dest_Queue[i] = {}
                                    ∧ Dispatch_Seq[i] = {}
        eff PendingCommand[i] := nil;
        PubCount[i] := 0;
        Dest_Seq[i] := {};
        first_cancel_ok[i] := 0;
        last_cancel_ok[i] := Inf
    output write(i:I, v:V)
        pre PendingCommand[i] = nil
        eff PendingWriteValue[i] := v;
        first_write_commit[i] :=  now + WriteCommmitTime.lb;
```

Figure 5.10 Specification for Change-Trigger automaton.

```
    last_write_commit[i] :=  now + WriteCommmitTime.ub;
    PendingCommand[i] := Commit
internal WriteCommit(i:I)
    pre PendingCommand[i] = Commit ∧ now ≥ first_write_commit[i] ∧ Generating =nil
    eff pub_value := [PendingWriteValue[i], pub_value.SeqNo + 1];
    WriteLog := WriteLog ⊢ [pub_value, i, now];
    PendingCommand[i] := Write;
    Generating:= i;
    first_generate[i] := now + generate_bound.lb;
    last_generate[i] := now + generate_bound.ub;
    first_write_ok[i] := Inf; %Inhibit write return until generationis done
    last_write_ok[i] := Inf;
    first_write_commit[i] :=0;
    last_write_commit[i] := Inf
internal GenerateMessages(i:I)
    pre Generating = i ∧ now ≥ first_generate[i]
    eff for j:I so that Subscribed[j] do
        Dispatch_Seq[j] := Dispatch_Seq[j] ⊢
              [pub_value, [now + dispatch_bound.lb, now + dispatch_bound.ub]]
    od;
    Generating := NULL;
    first_generate[i] := 0;
    last_generate[i] := Inf;
    first_write_ok[i] := now + write_ok.lb;
    last_write_ok[i] := now + write_ok.ub;
    PendingCommand[i] := Write
output write_ok(i:I)
    pre PendingCommand[i] = Write ∧ now ≥ first_write_ok[i]
    eff PendingCommand[i] := nil;
    first_write_ok[i] := 0;
    last_write_ok[i] := Inf
time-passage ν(t: Real)
    pre (∀ i:I ((now + t) < last_write_ok[i]
        ∧ (now + t) < last_cancel_ok[i]
        ∧ (now+ t) < last_sub_ok[i]
        ∧ ((len(Dispatch_Seq[i]) > 0) ⇒
           ((now + t) < head(Dispatch_Seq[i]).Delivery.ub))
        ∧ (len(Dest_Queue[i]) > 0 ⇒
           ((now + t) < head(Dest_Queue[i]).Delivery.ub))))
    eff now := now + t
```

Figure 5.10 Specification for Change-Trigger automaton.

- Case $\pi = WriteCommit_i$. This actions sets the automaton status, $PendingCommand_i = generate$. Thus the invariant is satisfied in s'.
- Case $\pi = GenerateMessages_i$. The effect of this action is to append an element T, where $T.X = tail(WriteLog).X$, to each $DispatchSeq_i$ where $subscribed_i$ holds. This automatically satisfies the invariant.
- Case $\pi = publish_i(T)$. This action transfers messages from the dispatch queue to the message channel. If W matches the head of $DispatchSeq_i$ then the invariant holds since we add T to $DestQueue_i$. If the element

```
invariant I1 of Publish_on_change:
   ∀ W: WriteRec (W ∈ WriteLog ⇒ (W.i = null ⇔ W = head(WriteLog)))

invariant I2 of Publish_on_change:
   ∀ R: RcvRec ((∀ i:I ((R ∈ Dest_Seq[i]) ⇒
                (∃ W:WriteRec (W ∈ WriteLog ∧ W.X.data = R.X.data)))))

invariant I3 of Publish_on_change:
   ∀ R: RcvRec (∀ i:I ((R ∈ Dest_Seq[i]) ⇒
                (∃ W:WriteRec (W ∈ WriteLog ∧ W.X = R.X))))

invariant I4 of Publish_on_change:
   ∀ i:I (∀ m: Int (∀ n: Int ((m ≥ 0 ∧ m < len(Dest_Seq[i])
                              ∧ n ≥ 0 ∧ n < len(Dest_Seq[i])
                              ∧ m < n) ⇒ Dest_Seq[i][m].X.SeqNo ≤ Dest_Seq[i][n].X.SeqNo)))

invariant I5 of Publish_on_change:
   ∀ t1:Real (∀ i:I (∀ R:RcvRec ((t1 = now ∧ R.t = t1 ∧ R ∈ Dest_Seq[i]) ⇒
              (pub_value = R.X ∨
              (head(WriteLog).X ≠ R.X ∧ head(WriteLog).t ≥ now - xmit.ub)))))

invariant I6 of Publish_on_change:
   ∀ i:I (now ≥ StartTime[i])

invariant I7 of Publish_on_change:
   ∀ i:I (PubCount[i] = len(Dest_Seq[i]))

invariant I8 of Publish_on_change:
   ∀ i:I ((len(Dest_Queue[i]) ≠ 0) ⇒ subscribed[i])

invariant I9 of Publish_on_change:
   ∀ i:I (∀ W:WriteRec (W ∈ WriteLog ∧ subscribed[i] ⇒ PendingCommand[i] = Generate ∨
                                ((∃ T:TxRec (T ∈ Dispatch_Seq[i] ∧ T.X = W.X))
                                 ∧ (∃ T:TxRec (T ∈ Dest_Queue[i] ∧ T.X = W.X))
                                 ∧ (∃ R:RcvRec (R ∈ Dest_Seq[i] ∧ T.X = W.X))))))

invariant I10a of Publish_on_change:
   ∀ i:I (∀ W:WriteRec (W ∈ WriteLog ⇒ ((∃ T:TxRec (T ∈ Dispatch_Seq[i] ∧ T.X = W.X)) ⇒
                                        now ≤ (W.t + dispatch_bound.ub))))

invariant I10b of Publish_on_change:
   ∀ i:I (∀ W:WriteRec (W ∈ WriteLog ⇒ ((∃ T:TxRec (T ∈ Dest_Queue[i] ∧ T.X = W.X)) ⇒
                                        (now ≥ (W.t + dispatch_bound.lb)
                                        ∧ now ≤ (W.t + dispatch_bound.ub + xmit.ub )))))

invariant I10c of Publish_on_change:
   ∀ i:I (∀ W:WriteRec (W ∈ WriteLog ⇒ ((∃ R:RcvRec (R ∈ Dest_Seq[i] ∧ R.X = W.X)) ⇒
                                        (R.t ≥ (W.t + dispatch_bound.lb + xmit.lb)))))
```

Figure 5.11 State invariants for Change-Triggered automata.

transferred is not the matching one, then the assertion is also clearly true from the inductive hypothesis.

- Case $\pi = pub\_rcv_i(T)$. This action satisfies the assertion for similar reasons to that of the *publish* action. The action either transfers the matching element to $DestSeq_i$, in which case a different clause of the invariant is satisfied, or it transfers a different element, in which case the assertion is trivially true.

Lemma 5.2.2: *In every reachable state of publish_on_change, the dispatches spend a bounded amount of time in* $DispatchSeq_i$ *and* $DestQueue_i$ *and must reach the client within dispatch_bound.ub + xmit.ub time units.*

Proof: This lemma concerns the assertions made in invariants I10a, I10b and I10c. For each invariant, we shall consider a separate induction proof, but since the results are similar, we combine the results into a single lemma. The start conditions for all three invariants trivially satisfy the assertion, since $subscribed_i = false$ for all $i \in I$. Let us consider the inductive step for I10a. We assume that the assertion holds for s and consider the transition (s, π, s'). All actions save for $subscribe(i)$, $Commit_i$, $publish(i, T)$, and $\nu(t)$ satisfy the assertion vacuously. The remaining actions we consider individually.

- Case $\pi = subscribe_i$. Since it is clear from the pseudocode that $\neg\exists W \in WriteLog, W.t > now$, it follows that there are no elements that satisfy the implication's predicate, i.e. $\neg\exists W \in WriteLog, W.t > StartTime_i$. Thus, the implication is trivially satisfied and the invariant holds in s'
- Case $\pi = WriteCommit_i$. The inductive hypothesis ensures that for all elements, W already in $WriteLog$ in state s, the assertion must also hold in s'. In addition, we must consider the extra element W' added to the $WriteLog$ by the *write* action. Since $W.t = now$, this element also satisfies the antecedent clause of the implication. Hence, I10a holds in s'.
- Case $\pi = publish_i(T)$ .This action removes elements from $DispatchSeq_i$. Therefore, in s', there are either the same or fewer matching elements in $DispatchSeq_i$ and the invariant must hold
- Case $\pi = \nu(t)$. It is clear that for elements T_1 and T_2 in $DispatchSeq_i$, that if $T_1 \preceq T_2$ then necessarily $T_1.bound \leq T_2.bound$. I.e. the nearest time-bound is at the front of the queue. For $\nu(t)$ to be enabled in s, $\forall i \in I, now + t \leq head(DispatchSeq_i).bound.ub$. Thus, the assertion must hold in s'.

The inductive step of the proof of I10b, proceeds in a similar fashion. We assume that the assertion holds for s, and consider the transition (s, π, s'). The non-trivial actions are $publish_i(T)$, $pub\_rcv_i(T)$, and $\nu(t)$.

- Case $\pi = publish_i(T)$.We consider the effect of the element, T, that we transfer from $DispatchSeq_i$ to $DestQueue_i$. For this action to be enabled, the first clause of the conjunction must hold. Also, from I10a, and the effect

of this action, it is clear that the second conjunctive clause must also hold. Thus, I10b holds in s'.

- Case $\pi = pub\_rcv_i(T)$. Since, we only remove elements from $DestQueue_i$ with this action, the invariant is unaffected.
- Case $\pi = \nu(t)$. Invariant I10a, gives an upper bound on the time spent in $DispatchSeq_i$. Let W_i' represent the $WriteLog$ element corresponding to the head element in the $DestQueue_i$. For $\nu(t)$ to be enabled $s.now + t \leq head(DestQueue_i).bound.ub$. We know that this value obeys $head(DestQueue_i).bound.ub \leq W_i'.t + dispatch\_bound.ub + xmit.ub$. Thus, the invariant is satisfied in s'.

Lastly, we consider the arrival of the dispatch messages at the user interface. The non-trivial actions for the inductive step of the proof are $pub\_rcv_i(T)$ and $cancel\_ok(i)$.

- Case $\pi = cancel\_ok_i$. The effect of this action is to erase the user record. Thus the assertion is vacuously true.
- Case $\pi = pub\_rcv_i(T)$. For this action to be enabled, $now \geq T.bound.lb$. From I10b and the effect of $publish$, if W' is the matching element of $WriteLog$ then $T.bound.lb \geq W'.t + dispatch\_bound.lb + xmit.lb$. This satisfies the invariant.

Theorem 5.2.1: *Any write invocation of the publish_on_change automaton at time* t_0, *results in delivery of exactly one publish message to each subscribed client within the time window* $t_0 + dispatch\_bound.lb + xmit.lb \leq now \leq t_0 + dispatch\_bound.ub + xmit.ub$.

Proof: This theorem follows directly from the previous assertions. Lemma 5.2.1 established the uniqueness of the notification, while Lemma 5.2.2 proves the time bounds. □

This completes the formal analysis of the GRRDE service specifications. We now examine the algorithms used in the actual GRRDE software and show that they implement these formalized services.

5.3 Analysis of GRRDE Implementation

The preceding section introduced the formal specification of the GRRDE runtime services and proved essential properties of their operation. We now provide a more concrete view of the software running in the GRRDE system. Our goal is to present the algorithms used

in the source code, and to show, using paired-simulation techniques, that they implement the abstract services described above. The concrete *publish* and *publish_on_change* automata are developed through the composition of several primitive automata (Figure 5.12, Figure 5.13). We first examine the functions performed by these components and then present the simulation proofs that relate the implementations to the specifications.

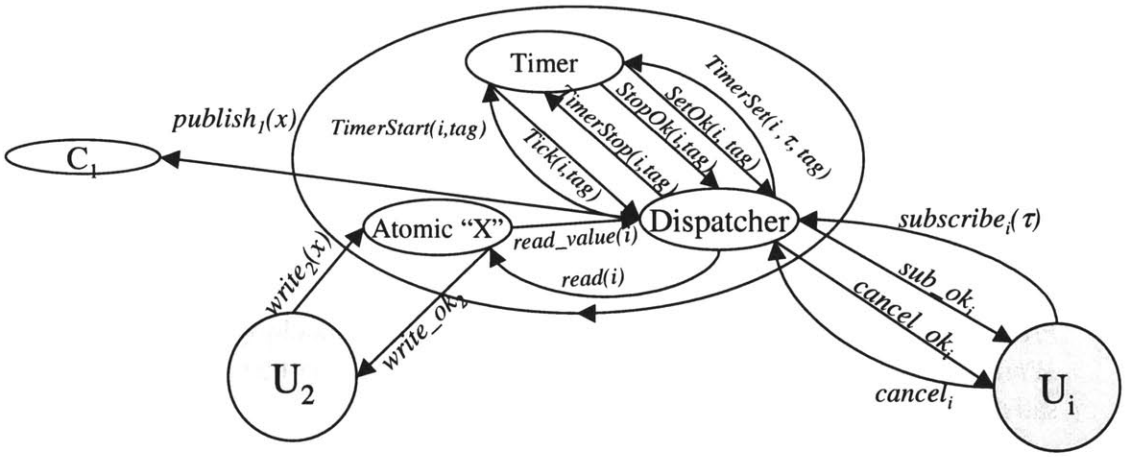


Figure 5.12 Composition of the *publish* automaton.

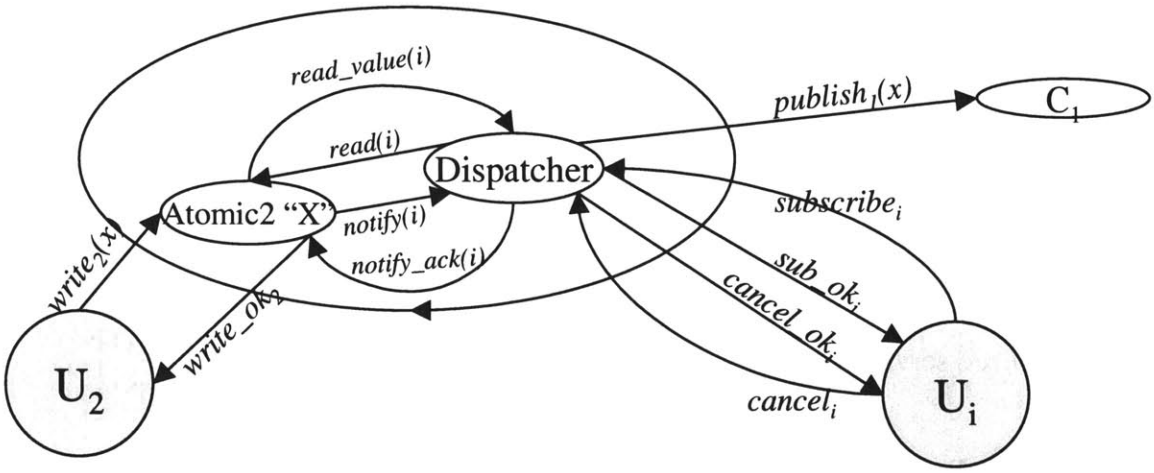


Figure 5.13 Composition of the *publish_on_change* automaton.

5.3.1 Atomic Objects

One of the primary building blocks of many distributed systems is the atomic object. Atomic objects ensure consistent, concurrent access to a variable shared by a number of processes. In the GRRDE system, an atomic object can only be accessed by processes on a single CPU. Thus, while GRRDE atomic objects must cope with concurrent access, they can assume a shared memory, accessible to all the system automata. Distributed atomic objects, in contrast, must include extra complexity to manage the mirroring of the state information between different physical locations.

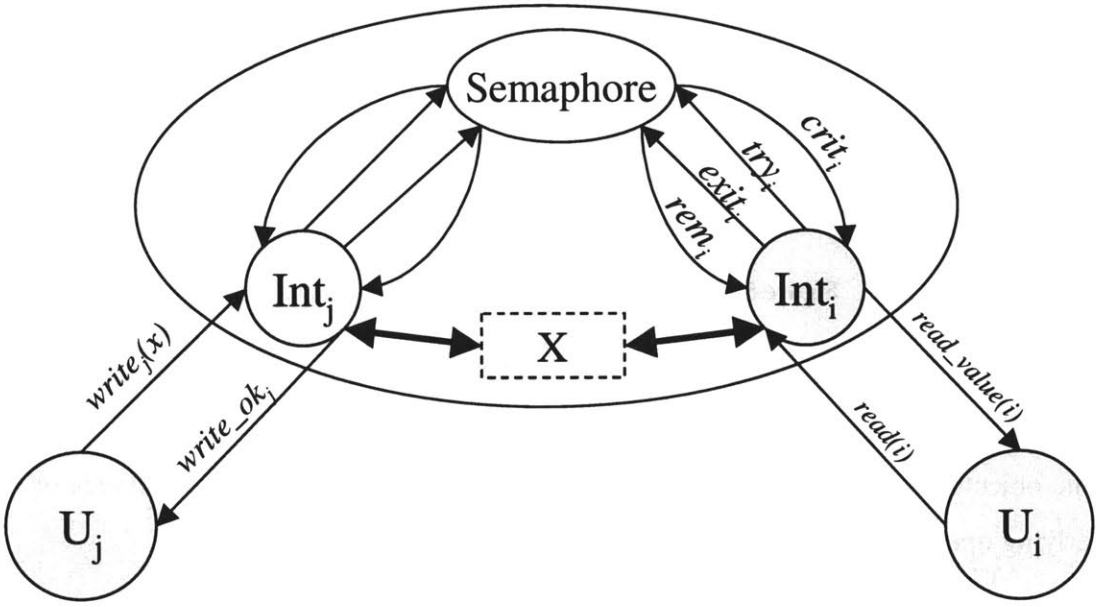


Figure 5.14 Composition of *atomic* automaton.

We have developed several models of GRRDE atomic objects. There are two primary varieties *AbstractAtomicVar* (Figure 5.14) and *AbstractAtomicVar2* (Figure 5.15). These automata are used in the *publish* and *publish_on_change* services, respectively. Since atomic objects are extremely important to the correct functioning of the publish-subscribe services, we present another layer of formal modelling. In the same way in which the abstract *publish* service is a composition of several automata, the

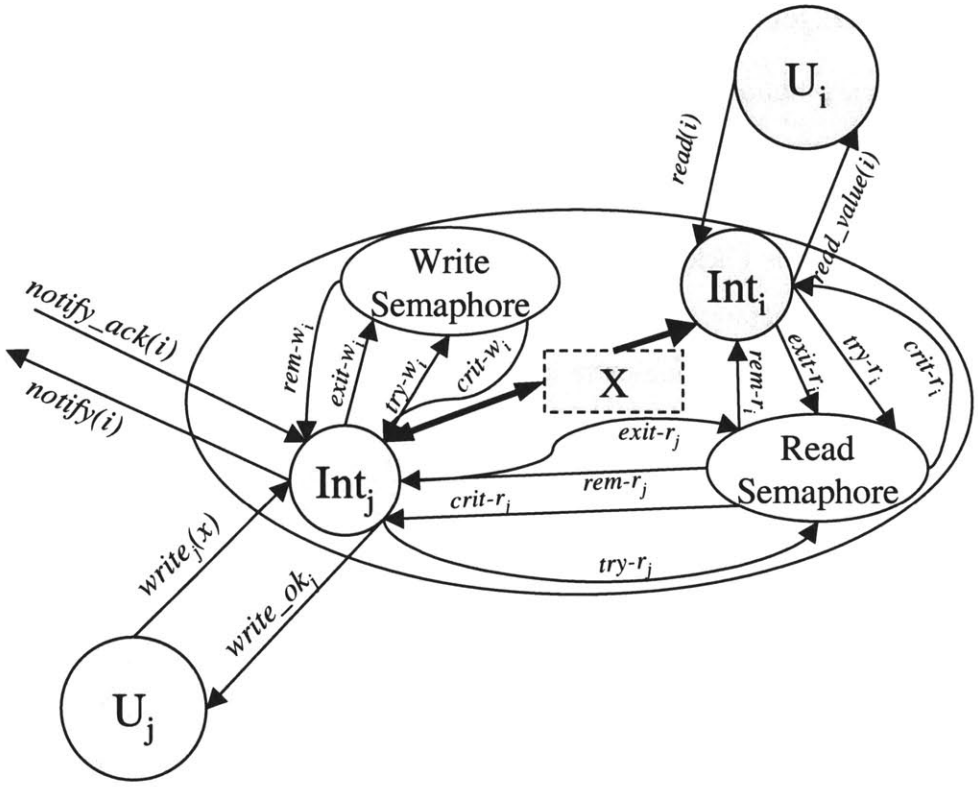


Figure 5.15 Composition of the *atomic2* automata.

AbstractAtomicVar automata can also be modelled hierarchically. First, we present an abstract specification for the atomic objects. Having established the basic properties of the atomic objects, we then show how these are achieved using the primitive services of the underlying operating system.

Let us start by examining the basic *AbstractAtomicVar* automaton. This automaton allows safe, asynchronous, read and write access to a shared variable of arbitrary type. In an ideal system, reading or writing to a memory location would be instantaneous. In reality, these operations may take a finite amount of time. For instance, when an automaton tries to read the atomic object, the request is made by a $read_i$ action. Some time later, a value is returned with a $return\_value_i(x)$ event. Similar invocations and responses are employed for $write_i(v)$ actions. When an object is shared by several processes or automata, the operations may overlap. These overlapping invocations must be handled in a man-

ner that allows the internal behaviour to be sensibly reconciled with the system's external behaviour. There must exist an instant, between an action's invocation and response, at which we can say that the operation took place. Making such assignments creates a total ordering of the invocations. This process is called *serializing*. The safety property that we require is that the operations performed on an atomic variable be *serializable* [Lynch, 1996]. It is not necessary that these instants be unique; we only require that some total ordering must exist.

The IOA pseudocode for the *AbstractAtomicVar* automaton is shown in Figure 5.16. This automaton supports concurrent interactions from a number of user processes. We record the current value of X in a *DataRec* tuple described in the previous section. The automaton maintains two log variables. The first is *WriteLog*, a list of all the values written to the object. We encountered this variable earlier, in the specification of the *publish* automata. Later, we will show the equivalence between the two instances. The other variable of interest is *Record*. This variable maintains a totally-ordered and time-tagged list of the *read* and *write* invocations. Entries are added to the list during the internal *do_read* and *do_write* actions.

Similarly, the IOA pseudocode for the *AbstractAtomicVar*2 automaton is shown in Figure 5.17. The response to $read_i$ invocations is similar to the preceding automaton. The $write_i(v)$ behaviour is slightly different. Each time that some automaton writes a value to this object we generate a $notify(i)$ action. We will show that this action is used to generate the publish dispatches. The *atomic*2 automaton must wait for a confirmatory $notifyAck(i)$ action before resuming normal operations and issuing a write-response. During the time in which an notification is pending, read invocations are permitted but write invocations are deferred. The *AbstractAtomicVar*2 automata maintain *WriteLog* and *Record* variables in a manner similar to the *AbstractAtomicVar* automata. One small change should be noted. In addition to recording the read and write events, this automaton also records the notify periods.

```
type ReadReq = tuple of i:I, bnd: TimeBound
type WriteReq = tuple of i:I, v: DataType, bnd: TimeBound
type SimplePC = enumeration of read_wait, read_done, write_wait, write_done, idle

automaton AbstractAtomicVar(v0: V, Inf: Real)
    signature
        %Interface to user
        input
            read(i:I),
            write(i:I, v:V)
        output
            read_value(i:I, x:DataRec),
            write_ok(i:I)
        time-passage
            ν(t:Real)
        internal
            do_read(i:I),
            do_write(i:I)
    states
        Record: Seq[OpRec] := {},
        value: DataRec := [v0,0],
        pc: Array[I, SimplePC] := constant(idle),
        PendingReadValue: Array[I, DataRec] := constant([v0,0]),
        PendingWriteValue: Array[I, DataRec] := constant([v0,0])
        %Time constants
        read_time : TimeBound,
        write_time : TimeBound,
        read_resp : TimeBound,
        write_resp : TimeBound,
        %Time accounting
        now : Real := 0,
        bnd: Array[I, TimeBound] := constant[0,Inf],
        first_read_resp :Array[I,Real] := constant(0),
        last_read_resp : Array[I, Real] := constant(Inf),
        first_write_resp : Array[I, Real] := constant(0),
        last_write_resp : Array[I, Real]  := constant(Inf),
        WriteLog: Seq[WriteRec] := {} ⊢ [[v0,0],NULL,0]
    transitions
        input read(i:I)
            eff bnd[i]:=[now + read_time.lb, now + read_time.ub];
            pc[i] := read_wait
        output read_value(i:I, x:DataRec)
            pre (pc[i] = read_done ∧ now ≥ first_read_resp[i] ∧ x = PendingValue[i])
            eff pc[i] := idle;
            first_read_resp[i] := 0;
            last_read_resp[i] := Inf
        input write(i:I, v: DataType)
            eff bnd[i] := [now+ write_time.lb, now + write_time.ub ];
            PendingWriteValue[i] := [v,0]; %SeqNo to be filled in later
```

Figure 5.16 IOA Code for *AbstractAtomicVar* automaton specification.

```
    pc[i] := write_wait
output write_ok(i:I)
    pre (pc[i] = write_done ∧ now ≥ first_write_resp[i])
    eff pc[i] := idle;
    first_write_resp[i] := 0;
    last_write_resp[i] := Inf
internal do_read(i:I)
    pre (pc[i] = read_wait ∧ now ≥ bnd[i].lb)
    eff pc[i] := read_done;
    PendingReadValue[i] := value;
    for j:I do
        if pc[j] = read_wait then
            bnd[j].lb = now + read_time.lb
        fi
    od;
    bnd[i] := [0,Inf];
    first_read_resp[i] := now + read_resp.lb;
    last_read_resp[i] := now + read_resp.ub;
    Record := Record ⊢ [read_end, i, now]
internal do_write(i:I)
    pre (pc[i] = write_wait ∧ now > bnd[i].lb)
    eff value := [PendingWriteValue[i].v, value.SeqNo + 1];
    pc[i] := write_done;
    for j:I do
        if pc[j] = write_wait then
            bnd[j].lb = now + write_time.lb
        fi
    od;
    bnd[i] := [0,Inf];
    first_write_resp[i] := now + write_resp.lb;
    last_write_resp[i] := now + write_resp.ub;
    Record := Record ⊢ [write_end, i, now];
    WriteLog := WriteLog ⊢ [value,i,now]
time-passage ν(t:Real)
    pre ∀ i:I ((now + t) < bnd[i].ub ∧ (now + t) < last_write_resp[i]
                                     ∧ (now + t) < last_read_resp[i])
    eff now := now + t
```

Figure 5.16 IOA Code for *AbstractAtomicVar* automaton specification.

We have not specified explicit invariants for either of these automata. Rather, we are using these specifications as an intermediate step in our overall formal analysis. Recall that we have presented the abstract publish services and are now examining their composition. Atomic objects are one of these components. However, the implementation of the atomic objects is sufficiently complex that the direct composition of the publish automaton would be quite confusing. Hence, the simple, abstract atomic objects will be used in the publish composition while their detailed implementation is described below.

```
type ReadReq = tuple of i:I, bnd: TimeBound
type WriteReq = tuple of i:I, v: DataType, bnd: TimeBound
type AbsWritePC = enumeration of write_wait, write_done, idle, notify, notify_done, Idle
type AbsReadPC = enumeration of read_wait, read_done, idle

automaton AbstractAtomicVar2(v0: V, Inf: Real)
    signature
        %Interface to user
        input
            read(i:I),
            write(i:I, v:V),
            notify_ack(i:I)
        output
            read_value(i:I, x:DataRec),
            write_ok(i:I),
            notify(i:I)
        time-passage
            ν(t:Real)
        internal
            do_read(i:I),
            do_write(i:I)
    states
        Record: Seq[OpRec] := {},
        WriteLog : Seq[WriteRec] := {} ⊢ [[v0,0], NULL, 0],
        value: DataRec := [v0,0],
        pc_read: Array[I, AbsReadPC] := constant(idle),
        pc_write: Array[I,AbsWritePC] := constant(idle),
        PendingWriteValue: Array[I, DataRec] := constant([v0,0]),
        PendingReadValue: Array[I,DataRec] := constant([v0,0]),
        Notifying: Bool := false,
        %Time constants
        read_time : TimeBound,
        write_time : TimeBound,
        read_resp : TimeBound,
        write_resp : TimeBound,
        notify_time: TimeBound,
        %Time accounting
        now : Real := 0,
        bnd: Array[I, TimeBound] := constant[0,Inf],
        first_read_resp :Array[I,Real] := constant(0),
        last_read_resp : Array[I, Real] := constant(Inf),
        first_write_resp : Array[I, Real] := constant(0),
        last_write_resp : Array[I, Real]  := constant(Inf)
    transitions
        input read(i:I)
            eff bnd[i]:=[now + read_time.lb, now + read_time.ub];
            pc_read[i] := read_wait
        output read_value(i:I, x:DataRec)
            pre (pc[i] = read_done ∧ now ≥ first_read_resp[i] ∧ x = PendingReadValue[i])
```

Figure 5.17 IOA Code for *AbstractAtomicVar2* automaton specification.

```
    eff pc[i] := idle;
    first_read_resp[i] := 0;
    last_read_resp[i] := Inf
input write(i:I, v:V)
    eff bnd[i] := [now+ write_time.lb, now + write_time.ub];
    PendingWriteValue[i] := [v,0];
    pc_write[i] := write_wait
output write_ok(i:I)
    pre (pc_write[i] = notify_done ∧ now ≥ first_write_resp[i])
    eff pc_write[i] := idle;
    first_write_resp[i] := 0;
    last_write_resp[i] := Inf
internal do_read(i:I)
    pre (pc_read[i] = read_wait ∧ now ≥ bnd[i].lb)
    eff pc_read[i] := read_done;
    PendingReadValue[i] := value;
    bnd[i] := [0,Inf];
    first_read_resp[i] := now + read_resp.lb;
    last_read_resp[i] := now + read_resp.ub;
    Record := Record ⊢ [read_end, i, now]
internal do_write(i:I)
    pre (pc[i] = write_wait ∧ now ≥ bnd[i].lb ∧ ¬Notifying)
    eff value := [PendingWriteValue[i].v, value.SeqNo];
    pc_write[i] := write_done;
    bnd[i] := [now + notify_time.lb ,now + notify_time.ub];
    Record := Record ⊢ [write_end, i, now];
    Record := Record ⊢ [notify_start, i, now];
    WriteLog := WriteLog ⊢ [value, i, now];
    Notifying := true
output notify(i:I)
    pre pc_write[i] = write_done ∧ now ≥ bnd[i].lb
    eff pc_write[i] := notify;
    bnd[i] := [0, Inf]
input notify_ack(i:I)
    eff pc_write[i] := notify_done;
    first_write_resp[i] := now + write_resp.lb;
    last_write_resp[i] := now + write_resp.ub;
    Record := Record ⊢ [notify_end, i, now];
    Notifying := false;
    for i:I do
        if pc_write[i] = write_wait then
            bnd[i].lb := now + write_time.lb
        fi
    od
time-passage ν(t:Real)
     pre ∀ i:I ((now + t) < bnd[i].ub ∧ (now + t) < last_write_resp[i]
                                      ∧ (now + t) < last_read_resp[i])
     eff now := now + t;
```

Figure 5.17 IOA Code for *AbstractAtomicVar2* automaton specification.

The key to constructing an atomic object in a shared memory system is ensuring exclusive access to the variable storage during the time in which the variable is accessed. For simple

variables such as integer or floats, memory access in most processors is inherently atomic. Unfortunately, for more complex variables, such as arrays or structures, this is not the case. A read that overlaps with a write, may return partly old values and partly new values. This *inconsistency* is extremely undesirable. To ensure atomicity, the processes sharing the variable must execute a *mutual exclusion (ME)* algorithm. The canonical representation of such a protocol involves four actions:

1. try_i, in which a process announces its intention to compete for the shared resource.
2. $crit_i$, in which the ME algorithm grants exclusive access to the resource.
3. $exit_i$, in which a process currently accessing the resource, releases exclusive control
4. rem_i, in which a releasing process is given confirmation of the resource release.

GRRDE employs operating system *semaphores* to achieve mutual exclusion.

Semaphores are special types of variables that are accessed through operating system functions. They are initialized with a starting integer value. This value represents the number of process allowed to access the resource at once (i.e. one, for mutual exclusion). Processes may use try_i to attempt to secure the semaphore. The current value is checked. If the semaphore value is greater than zero, the value is decremented and the process may proceed. If the value is zero, the process is blocked. When finished, the $exit_i$ action increments the counter and allows another process to access the resource. Although, it is not a universal stipulation of ME, the OSE operating system actually guarantees FIFO response to try_i requests, i.e the waiting processes are queued. The behaviour of OSE semaphores can be described abstractly by an IOA model (Figure 5.18). The state invariants for the $SemaphoreMutEx$ automaton are given in Figure 5.19.

Lemma 5.3.1: *In every reachable state of* $SemaphoreMutEx$*, either if* $status_i = crit$ *then* $active = i$ *and the semaphore value is less than 1.*

```
type region = enumeration of try, crit, exit, rem
type TimeBound = tuple of lb: Real, ub: Real

automaton SemaphoreMutEx(Inf:Real)
    signature
        input try(i:I), exit(i:I)
        output crit(i:I), rem(i:I)
        time-passage ν(t:Real)
    states
        %"Constant"
        d:Real := 1,
        sem_queue:Seq[I],
        status: Array[I,region] := constant(rem),
        sem_value: Int := 1,
        active : I,
        now : Real := 0,
        Rem: Array[I,TimeBound] := constant([0,Inf])
    transitions
        input try(i:I)
            eff sem_queue := sem_queue ⊢ i;
            status[i] := try
        output crit(i:I)
            pre len(sem_queue) ≠ 0 ∧ head(sem_queue) = i ∧ sem_value > 0
            eff sem_value := sem_value - 1;
            active := i;
            sem_queue:= tail(sem_queue);
            status[i] := crit
        input exit(i:I)
            eff status[i] := exit;
            Rem[i].ub := now + d;
            sem_value := sem_value + 1
        output rem(i:I)
            pre status[i] = exit
            eff status[i] := rem;
            Rem[i].ub := Inf
        time-passage ν(t:Real)
            pre ∀ i:I ((now + t) > Rem[i].ub ∧ (sem_value = 0 ∨ status[i] = rem ∨ status = exit))
            eff now := now + t
```

Figure 5.18 The *SemaphoreMutEx* automaton.

```
invariant I1 of SemaphoreMutEx:
  ∀ i:I ((status[i] = crit) ⇒ (∀ j:I (j=i ∨ status[j]≠crit)))

invariant I2a of SemaphoreMutEx:
  sem_value = 0 ∨ sem_value = 1

invariant I2b of SemaphoreMutEx:
  ∀ i:I ((status[i] = crit) ⇒ (active = i ∧ sem_value < 1))
```

Figure 5.19 Invariants of the *SemaphoreMutEx* automaton.

Proof: We prove I2b, by induction on the states of *SemaphoreMutEx*. The invariant clearly holds in the start state. Proceeding to the inductive step, we assume that the assertion is true in state s and consider the transition (s, π, s'). The actions $\nu(t)$ and rem_i are vacuously true. We must consider the remaining actions individually:

- Case $\pi = try_j$. If $status_i = crit$, then since we assume that user interactions are well formed, it is necessarily the case that $i \neq j$, i.e it was a different process trying to access the resource. The assertion must therefore hold in s'. If $status_i \neq crit$, then the assertion is trivially satisfied.
- Case $\pi = crit_j$. If $s.status_i = crit$, then from the invariant assertion, this action cannot be enabled. If $s.sem\_value = 1$, then this action could be enabled. Thus $s'.status_j = crit$, $s'.sem\_value = 0$, and $s'.active = j$ and the assertion is satisfied.
- Case $\pi = exit_j$. From the invariant assertion and the assumption of well-formedness, if $s.status_i = crit$ then $i = j$. Thus, the assertion is trivially true in s'.

Lemma 5.3.2: *In every reachable state of SemaphoreMutEx, only one process may be in the critical region at a time.*

Proof: The property of assertion I1 can be established with a simple induction on the states of *SemaphoreMutEx*. The condition is clearly true in the automaton start state. Thus, proceed with the inductive step. Assuming that the assertion holds in state s, we must consider the transition (s, π, s') and prove that it holds in state s'. The actions in which the assertion is not vacuously true are $crit_i$ and $exit_i$. All other actions cannot affect the invariant.

- Case $\pi = crit_j$. If $\neg\exists i, s.status_i = crit$, then the assertion will hold in s'. Otherwise, from Lemma 5.3.1, this action cannot be enabled.
- Case $\pi = exit_j$. From the invariant hypothesis and the effects of this action we know that $\neg\exists i, s'.status_i = crit$. Thus, the assertion must hold in s'.

Theorem 5.3.1: *The SemaphoreMutEx ensures mutual exclusion.*

Proof: Since from Lemma 5.3.2, there cannot be more than one user with $status_i = crit$, exclusive access is guaranteed. □

The *atomic* automaton is constructed by composing one *SemaphoreMutEx* automaton a number of interface automata, $AtomicVarInterface_i$ (Figure 5.14) and the shared variable X. The various $AtomicVarInterface_i$ automata manage the atomic access to the variable X and present a distributed interface to the user processes. The IOA model for $AtomicVarInterface_i$ is given in Figure 5.20. The state invariants are found in Figure 5.21. Although the IOA code represents a number of distributed automata, not all the state information is split up. The artificial log variables *Record* and *WriteLog* are shared, as is the shared variable X. The operation of the composed automata is straightforward. After receiving a $read_i$ or $write_i(v)$ invocation, the interface automata dispatches a try_i action to the semaphore. Once it receives permission to proceed ($crit_i$), the interface will perform the requested operation on X and then issue an $exit_i$ to release the lock. The final rem_i message from the semaphore permits the operation to complete and the interface produces a $read\_value_i(x)$ or $write\_ok_i$ message.

We combine the automata $AtomicVarInterface_i$ with a *SemaphoreMutEx* to produce the composed automaton $AtomicVar$. We can prove several state invariants about the composition.

Lemma 5.3.1: *In every reachable state of $AtomicVar_i$ at most one process is reading or writing at a given time.*

Proof: We prove I3 through induction on the states of $AtomicVar_i$. Initially, the assertion is clearly true. To establish the inductive step we assume that I3 holds in state s and consider the transition to state s', i.e. (s, π, s'). The only actions for which the assertion is not vacuously true are $crit_i$ and $exit_i$.

- Case $\pi = crit_i$. If there no writing or reading processes in state s, the assertion will be satisfied in s' since the status of only one user changes. If there is an i such that $s.pc_i = reading \vee s.pc_i = writing$, from Theorem 5.3.1, we cannot receive this action since that would violate mutual exclusion. Thus the assertion is satisfied in s'
- Case $\pi = exit_i$. This action trivially satisfies the assertion since the predicate of the implication becomes false.

```
type Op = enumeration of read_start, read_end, write_start, write_end
type OpRec = tuple of op:Op, i:I, t:Real
type PC = enumeration of idle, write_enable,  write_block, writing, write_release,
            write_return, read_enable, read_block, reading, read_release, read_return

automaton AtomicVarInterface(v0 :V, Inf: Real)
    signature
        %Interface to user
        input
            read(i:I),
            write(i:I, v:V),
            crit(i:I), rem(i:I)
        output
            read_value(i:I, x:DataRec),
            write_ok(i:I),
            try(i:I), exit(i:I)
        time-passage
            ν(t: Real)
    states
        %We have one program counter for each automaton
        pc : Array[I, PC] := constant(idle),
        value: DataRec := [v0,0],
        PendingValue: Array[I, DataRec]:= constant([v0,0]),
        %TimeBound Constants
        r_dispatch_bnd: TimeBound,
        w_dispatch_bnd: TimeBound,
        r_execute_bnd: TimeBound,
        w_execute_bnd: TimeBound,
        r_return_bnd: TimeBound,
        w_return_bnd: TimeBound,
        %Time accounting
        now : Real := 0,
        first_read_dispatch: Array[I, Real] := constant(0),
        last_read_dispatch: Array[I, Real] := constant(Inf),
        first_write_dispatch: Array[I, Real] := constant(0),
        last_write_dispatch: Array[I, Real] := constant(Inf),
        first_read_exec: Array[I, Real] := constant(0),
        last_read_exec: Array[I, Real] := constant(Inf),
        first_write_exec: Array[I, Real] := constant(0),
        last_write_exec: Array[I, Real] := constant(Inf),
        first_write_return: Array[I, Real] := constant(0),
        last_write_return: Array[I, Real] := constant(Inf),
        %Action log
        Record : Seq[OpRec] := {},
        WriteLog : Seq[WriteRec] := {} ⊢ [[v0,0],NULL,0]
    transitions
        input read(i:I)
            eff pc[i] := read_enable;
            first_read_dispatch[i] := now + r_dispatch_bnd.lb;
```

Figure 5.20 *AtomicVarInterface* automata pseudocode.

```
    last_read_dispatch[i] := now + r_dispatch_bnd.ub
input write(i:I, v:V)
    eff pc[i] := write_enable;
    first_write_dispatch[i] := now + r_dispatch_bnd.lb;
    last_write_dispatch[i] := now + r_dispatch_bnd.ub;
    PendingValue[i] := [v,0] % sequence number assigned later
output try(i:I)
    pre ((pc[i] = read_enable ∧ now ≥ first_read_dispatch[i])
                ∨ (pc[i] = write_enable ∧ now ≥ first_write_dispatch[i]))
    eff if pc[i] = read_enable then
        pc[i] := read_block;
        first_read_dispatch[i] := 0;
        last_read_dispatch[i] := Inf
    else
        pc[i] := write_block;
        first_write_dispatch[i] := 0;
        last_write_dispatch[i] := Inf
    fi
input crit(i:I)
    eff if pc[i] = read_block then
        pc[i] := reading;
        first_read_exec := now + r_execute_bnd.lb;
        last_read_exec := now + r_execute_bnd.ub;
        Record := Record ⊢ [read_start,i, now]
    else
        pc[i] := writing;
        first_write_exec[i] := now + w_execute_bnd.lb;
        last_write_exec[i] := now + w_execute_bnd.ub;
        Record := Record ⊢ [write_start, i, now]
    fi
output exit(i:I)
    pre ((pc[i] = reading ∧ now ≥ first_read_exec[i])
                ∨ (pc[i] = writing ∧ now ≥ first_read_exec[i]))
    eff if (pc[i] = reading) then
        pc[i] := read_release;
        PendingValue[i] := value;
        first_read_exec[i] := 0;
        last_read_exec[i] := Inf;
        Record := Record ⊢ [read_end, i, now]
    else
        pc[i] := write_release;
        value := [PendingValue[i].v, value.SeqNo+1];
        first_write_exec[i] := 0;
        last_write_exec[i] := Inf;
        Record := Record ⊢ [write_end, i, now];
        WriteLog := WriteLog ⊢ [value,i,now]
    fi
input rem(i:I)
    eff if (pc[i] = read_release) then
       pc[i] := read_return;
```

Figure 5.20 *AtomicVarInterface* automata pseudocode.

```
        first_read_return[i] := now + r_return_bnd.lb;
        last_read_return[i] := now + r_return_bnd.ub
      else
        pc[i] := write_return;
        first_write_return[i] := now + w_return_bnd.lb;
        last_write_return[i] := now + w_return_bnd.ub
      fi
  output read_value(i:I, x:DataRec)
      pre (now ≥ first_read_return[i] ∧ x = PendingValue[i]
                                      ∧ pc[i] = read_return)
      eff pc[i] := idle;
      first_read_return[i] := 0;
      last_read_return[i] := Inf
   output write_ok(i:I)
       pre (now ≥ first_write_return[i] ∧ pc[i] = write_return)
       eff pc[i] := idle;
       first_write_return[i] := 0;
       last_write_return[i] := Inf
   time-passage ν(t:Real)
       pre (∀ i:I ((now+t) < last_read_return[i]
                   ∧ (now + t) < last_write_return[i]
                   ∧ (now + t) < last_write_exec[i]
                   ∧ (now + t) < first_read_exec[i]
                   ∧ (now + t) < last_read_dispatch[i]
                   ∧ (now+t) < last_write_dispatch))
       eff now := now + t
```

Figure 5.20 *AtomicVarInterface* automata pseudocode.

```
invariant I3 of AtomicVarInterface:
  ∀ i:I ((pc[i] = writing ∨ pc[i] = reading) ⇒
        (¬∃ j:I (i≠j ∧ (pc[j] = writing ∨ pc[j] = reading))))
invariant I4a of AtomicVarInterface:
  ∀ m:Int ((m≥0 ∧ m < len(Record) ∧ Record[m].op = read_start) ⇒
           (Record[m] = last(m)
           ∨ (Record[m+1].op = read_end ∧ Record[m].i = Record[m+1].i)))
invariant I4b of AtomicVarInterface:
  ∀ m:Int ((m ≥ 0 ∧ m < len(Record) ∧ Record[m].op = write_start) ⇒
          (Record[m] = last(m)
          ∨ (Record[m+1].op = write_end ∧ Record[m].i = Record[m+1].i)))
```

Figure 5.21 Invariants of *AtomicVarInterface*.

Lemma 5.3.2: *In every reachable state of* $AtomicVar_i$*, the logged read and write intervals do not overlap.*

Proof: For the sake of clarity we treat the *read* (I4a) and *write* (I4b) intervals separately. The automata create a *start* entry in *Record* when they receive a $crit_i$ message

and write an *end* entry when they send an $exit_i$ message. The invariants state that every *start* entry must be followed immediately by a corresponding *end* entry, or it must be the last element of the log. Since the two invariants are almost identical, we shall sketch out the proofs in parallel. We prove these properties by induction on the states of $AtomicVar_i$. The properties are clearly true in the start state, therefore we may proceed with the inductive step. Assuming that I4a and I4b are true in s, we consider their validity under the transition (s, π, s'). Only two actions affect $Record$:

- Case $\pi = crit_i$. If the previous interval has been 'closed' then, this action will add a *start_write* or *start_read* entry to $Record$. Since this new entry would be the last element of $Record$, the invariant will hold in s'. If the previous interval is 'open', this action cannot occur since it would contradict Lemma 5.3.1.
- Case $\pi = exit_i$. Since this action will close any open interval, the invariant is clearly true in state s'.

Theorem 5.3.1: *The $AtomicVar$ composition implements safe atomic access to variable X.*

Proof: Since the access to the variable X is controlled by a mutual exclusion protocol (Theorem 5.2.1) and the and each operation possess a non-overlapping 'execution' interval (Lemma 5.3.2), each operation can be serialized to any point during interval. Thus, the *atomic* automaton correctly implements an atomic object. □

A similar process can be followed to compose the $atomic2$ automaton. This composition requires two $SemaphoreMutEx$ automata as well as the interface automata $AtomicVarInterface2$. One of the semaphores provide a 'write-lock', and the other provides a 'read-lock'. To read from the object, a process needs to secure the read-semaphore. To write to the object a process must obtain both at the same time. Once the physical write operation is complete, the read-semaphore is released, but the write-semaphore is retained until the notification process is concluded. The IOA code for the $AtomicVarInterface2$ automata is given in Figure 5.22.

```
type Op = enumeration of read_start, read_end, write_start, write_end, notify_start, notify_end
type OpRec = tuple of op:Op, i:I, t:Real
type ReadPC = enumeration of idle, read_enable, read_block,
            reading, read_release, read_return
type WritePC = enumeration of idle, write_enable,  write_block, writing, write_release,
            write_return, notify_enable, notifying, notify_release, write_read_enable,
            write_read_block, write_read_release

automaton AtomicVarInterface2(v0 :V, Inf: Real)
    signature
        %Interface to user
        input
            read(i:I),
            write(i:I, v:V),
            crit_r(i:I), rem_r(i:I),
            crit_w(i:I), rem_w(i:I),
            notify_ack(i:I)
        output
            read_valu,e(i:I, x:DataRec),
            write_ok(i:I),
            try_r(i:I), exit_r(i:I),
            try_w(i:I), exit_w(i:I),
            notify_ack(i:I)
        time-passage
            ν(t: Real)
    states
        %We have one program counter for each automaton
        pc_read : Array[I, ReadPC] := constant(idle),
        pw_write: Array[I, WritePC] := constant(idle),
        value: DataRec := [v0,0],
        PendingReadValue: Array[I, DataRec] := constant([v0,0]),
        PendingWriteValue: Array[I, DataRec] := constant([v0,0]),
        %TimeBound Constants
        r_dispatch_bnd: TimeBound,
        w_dispatch_bnd: TimeBound,
        r_execute_bnd: TimeBound,
        w_execute_bnd: TimeBound,
        r_return_bnd: TimeBound,
        w_return_bnd: TimeBound,
        %Time accounting
        now : Real := 0,
        first_read_dispatch: Array[I, Real] := constant(0),
        last_read_dispatch: Array[I, Real] := constant(Inf),
        first_write_dispatch: Array[I, Real] := constant(0),
        last_write_dispatch: Array[I, Real] := constant(Inf),
        first_read_exec: Array[I, Real] := constant(0),
        last_read_exec: Array[I, Real] := constant(Inf),
        first_write_exec: Array[I, Real] := constant(0),
        last_write_exec: Array[I, Real] := constant(Inf),
```

Figure 5.22 The *AtomicVarInterface2* automata.

```
    first_write_return: Array[I, Real] := constant(0),
    last_write_return: Array[I, Real] := constant(Inf),
    %Action log
    Record : Seq[OpRec] := {0}
transitions
    input read(i:I)
        eff pc_read[i] := read_enable;
        first_read_dispatch[i] := now + r_dispatch_bnd.lb;
        last_read_dispatch[i] := now + r_dispatch_bnd.ub
    input write(i:I, v:V)
        eff pc_write[i] := write_enable;
        first_write_dispatch[i] := now + w_dispatch_bnd.lb;
        last_write_dispatch[i] := now + w_dispatch_bnd.ub;
        PendingWriteValue[i] := [v,0]
    output try_r(i:I)
        pre ((pc_read[i] = read_enable ∧ now ≥ first_read_dispatch[i])
                    ∨ (pc_write[i] = write_read_enable ∧ now ≥ first_write_dispatch[i]))
        eff if pc_read[i] = read_enable then
            pc_read[i] := read_block;
            first_read_dispatch[i] := 0;
            last_read_dispatch[i] := Inf
        else
            pc_write[i] := write_read_block;
            first_write_dispatch[i] := 0;
            last_write_dispatch[i] := Inf
        fi
    output try_w(i:I)
        pre pc_write[i] = write_enable ∧ now ≥ first_write_dispatch[i]
        eff pc_write[i] := write_block;
        first_write_dispatch[i] := 0;
        last_write_dispatch[i] := Inf
    input crit_r(i:I)
        eff if pc_read[i] = read_block then
            pc_read[i] := reading;
            first_read_exec := now + r_execute_bnd.lb;
            last_read_exec := now + r_execute_bnd.ub;
            Record := Record ⊢ [read_start,i, now]
        else
            pc_write[i] := writing;
            first_write_exec[i] := now + w_execute_bnd.lb;
            last_write_exec[i] := now + w_execute_bnd.ub;
            Record := Record ⊢ [write_start, i, now]
        fi
    input crit_w(i:I)
        eff pc_write[i] := write_read_enable;
        first_write_dispatch[i] := now + r_dispatch_bnd.lb;
        last_write_dispatch[i] := now + r_dispatch_bnd.ub;
    output exit_r(i:I)
        pre ((pc_read[i] = reading ∧ now ≥ first_read_exec[i])
                    ∨ (pc_write[i] = writing ∧ now ≥ first_read_exec[i]))
```

Figure 5.22 The *AtomicVarInterface2* automata.

```
    eff if (pc_read[i] = reading) then
        pc_read[i] := read_release;
        PendingReadValue[i] := value;
        first_read_exec[i] := 0;
        last_read_exec[i] := Inf;
        Record := Record ⊢ [read_end, i, now]
    else
        pc[i] := write_read_release;
        value := [PendingWriteValue[i].v, value.SeqNo + 1];
        first_write_exec[i] := 0;
        last_write_exec[i] := Inf;
        Record := Record ⊢ [write_end, i, now];
        Record := Record ⊢ [notify_start, i, now];
        WriteLog := WriteLog ⊢ [value, i, now];
    fi
output exit_w(i:I)
    pre pc_write[i] = notify_release ∧ now ≥ first_write_exec[i]
    eff pc_write[i] := write_release;
    first_write_exec[i] := 0;
    last_write_exec[i] := Inf;
    Record:= Record ⊢ [notify_end, i, now]
input rem_r(i:I)
    eff if (pc_read[i] = read_release) then
       pc_read[i] := read_return;
       first_read_return[i] := now + r_return_bnd.lb;
       last_read_return[i] := now + r_return_bnd.ub
    else
       pc_write[i] := notify_enable;
       first_write_return[i] := now + w_dispatch_bnd.lb;
       last_write_return[i] := now + w_dispatch_bnd.ub
    fi
input rem_w(i:I)
    eff pc_write := write_return;
    first_write_return[i] := now + w_return_bnd.lb;
    last_write_return[i] := now + w_return_bnd.ub;
output read_value(i:I, x:DataRec)
    pre (now ≥ first_read_return[i] ∧ x = PendingReadValue[i]
                                    ∧ pc_read[i] = read_return)
    eff pc_read[i] := idle;
    first_read_return[i] := 0;
    last_read_return[i] := Inf
output write_ok(i:I)
     pre (now ≥ first_write_return[i] ∧ pc[i] = write_return)
     eff pc_write[i] := idle;
     first_write_return[i] := 0;
     last_write_return[i] := Inf
 output notify(i:I)
     pre pc_write[i] = notify_enable ∧ now ≥ first_write_exec[i]
     eff pc_write[i] := notifying;
     first_write_exec[i] := 0;
```

Figure 5.22 The *AtomicVarInterface2* automata.

```
    last_write_exec[i] := Inf
input notify_ack(i:I)
    eff pc_write[i] := notify_release;
    first_write_exec[i] := now + w_return_bnd.lb;
    last_write_exec[i] := now + w_return_bnd.ub
time-passage ν(t:Real)
    pre (∀ i:I ((now+t) < last_read_return[i]
                ∧ (now + t) < last_write_return[i]
                ∧ (now + t) < last_write_exec[i]
                ∧ (now + t) < first_read_exec[i]
                ∧ (now + t) < last_read_dispatch[i]
                ∧ (now+t) < last_write_dispatch))
    eff now := now + t
```

Figure 5.22 The *AtomicVarInterface2* automata.

When the two semaphores are composed with the interface automata, some of their actions must be renamed to eliminate ambiguity. For instance the try_i action for the read-semaphore has been relabeled try_r_i. The invariants in can be proven from the composition. The composition is called $AtomicVar2_i$.

```
invariant I3 of AtomicVarInterface2:
   ∀ i:I ((pc_write[i] = writing ∨ pc_read[i] = reading) ⇒
          (¬∃ j:I (i≠j ∧ (pc_write[j] = writing ∨ pc_read[j] = reading))))

invariant I3a of AtomicVarInterface2:
   ∀ i:I ((pc_write[i] ∈
  {write_read_enable, write_read_block, write_read_release, notify_enable, notifying, notify_release}
    ⇒ (¬∃ j:I ( i≠ j ∧ pc_write[j] ∈
{write_read_enable, write_read_block, write_read_release, notify_enable, notifying, notify_release}))))

invariant I4a of AtomicVarInterface2:
   ∀ m:Int ((m≥0 ∧ m < len(Record) ∧ Record[m].op = read_start) ⇒
             (Record[m] = last(m)
             ∨ (Record[m+1].op = read_end ∧ Record[m].i = Record[m+1].i)))

invariant I4b of AtomicVarInterface2:
   ∀ m:Int ((m ≥ 0 ∧ m < len(Record) ∧ Record[m].op = write_start) ⇒
            (Record[m] = last(m)
            ∨ (Record[m+1].op = write_end ∧ Record[m].i = Record[m+1].i)))

invariant I4c of AtomicVarInterface2:
   ∀ S:Seq[OpRec] ((S ≼ Record ∧ head(S).Op = notify_start
                ∧ ((last(S) = last(Record) ∧ ¬∃ R:OpRec (R ∈ S ∧ R.op = notify_end))
                  ∨ (last(S).op = notify_end ∧ ¬∃ R:OpRec (R ∈ init(S) ∧ R.op = notify_end))))
                   ⇒ (¬∃ R:OpRec (R ∈ S ∧ (R.op = write_start ∨ R.op = write_end))))
```

Figure 5.23 Invariants of *AtomicVar2*.

Lemma 5.3.1: *In every reachable state of* $AtomicVar2$*, only one process may be reading or writing at the same time.*

Proof: To prove I3 we consider induction on the states of $AtomicVar2$. Clearly the invariant is true in the start state. We proceed with the inductive step. Assuming that the invariant holds in state s, we consider the transition (s, π, s'). The actions for which the invariant is not vacuously true are $crit\_r_i$ and $exit\_r_i$.

- Case $\pi = crit\_r_i$. Satisfaction of the invariant follows directly from the mutual exclusion guarantee of the read-semaphore. Either there is no reading or writing process in s, in which case the execution of this action changes the status of a single process, or this action cannot occur at all.
- Case $\pi = exit\_r_i$. This action makes the invariant trivially true in s' by invalidating the predicate to the implication.

Lemma 5.3.2: *In every reachable state* $AtomicVar2$*, once a process has secured the write-semaphore, further write operations are deferred until the end of the notification phase.*

Proof: Invariant I3a follows directly from an induction on the states of $AtomicVar2$. Clearly the invariant is true in the start state. We must then assume that the assertion holds in state s and examine the transition (s, π, s'). We need consider only actions $crit\_w_i$ and $exit\_w_i$. Although there are many transitions between these two events either the invariant hypothesis precludes them being enabled, or well-formedness assumptions about the $notify_i$ and $notify\_ack_i$ exchanges prohibits the event.

- Case $\pi = crit\_w_i$.The satisfaction of the invariant follows directly from the mutual exclusion guarantee of the write-semaphore. Either there is no reading or writing process in s, in which case the execution of this action changes the status of a single process, or this action cannot occur.
- Case $\pi = exit\_w_i$. This action makes the invariant trivially true in s' by invalidating the predicate to the implication.

Lemma 5.3.3: *In every reachable state of* $AtomicVar2$*, read and write intervals do not overlap.*

Proof: The proof of invariants I4a and I4b follows an inductive proof similar to Lemma 5.3.2. The assertion is clearly true in the start state. The non-vacuous actions are $crit\_r_i$ and $exit\_r_i$.

- Case $\pi = crit\_r_i$.The satisfaction of the invariant follows directly from the mutual exclusion guarantee of the read-semaphore. Either there is no reading or writing process in s, in which case the execution of this action changes the status of a single process, or this action cannot occur.
- Case $\pi = exit\_r_i$. This action will close an open operation interval. Thus the assertion becomes trivially true in s'.

Lemma 5.3.4: *In every reachable state of* $AtomicVar2$, *there are no write_start or write_end events recorded during a notify period.*

Proof: Invariant I4c follows from induction on the states of $AtomicVar2$. We must consider $crit\_r_i$, $exit\_r_i$, and $exit\_w_i$.

- Case $\pi = crit\_r_i$. If there is currently an open notify interval in $Record$, then only reading processes can execute this action. Writing processes are barred by Lemma 5.3.3. If there is no open notify interval, the assertion is trivially satisfied
- Case $\pi = exit\_r_i$. From the preceding case, we cannot open a write interval in a notify interval. Furthermore, we know that we close the write interval when we open the notify interval. Thus, only reading processes can execute this action in an open notify interval. The assertion is satisfied.
- Case $\pi = exit\_w_i$. This action will close a *notify* interval and trivially satisfy the invariant.

Theorem 5.3.1: *The* $AtomicVar2$ *composition implements safe, atomic access to variable* X.

Proof: This follows directly from the mutual exclusion properties of the semaphores (Theorem 5.2.1) and the serializable read and write operations (Lemma 5.3.3) in $Record$.□

Let us now examine the relation between the abstract and concrete versions of the atomic objects. We wish to prove that *AtomicVar* implements the specification provided in *AbstractAtomicVar*. To establish this property we consider a paired-simulation proof.

To establish a simulation relation between two automata, A and B, we must introduce a relation F between their respective states, s and u, such that $u \in F(s)$. This relation is proven inductively. It must hold in the start states of the automata. Furthermore, we must show that for each transition, (s, π, s') of A, there is some execution fragment β of B that brings $u \xrightarrow{\beta} u'$ such that $u \in F(s)$ and $u' \in F(s')$. The execution fragment, β, may consist of zero or more actions of B, but must have the same external trace as π.

The mapping function that relates the states s of *AtomicVar* to the states u of *AbstractAtomicVar* is given in Figure 5.24.

```
automaton AtomicVar(v0: DataType, Inf: Real)
    components Interface:AtomicVarInterface(v0, Inf)
               MutEx: SemaphoreMutEx(Inf),
          hidden: try(i), crit(i), rem(i), exit(i)

%Simple assertion of state consistency
invariant I5 of AtomicVar:
    ∀ i:I ((MutEx.Status[i] = crit) ⇔ (Interface.pc[i] = writing ∨ Interface.pc[i] = reading))

forward simulation from AtomicVar to AbstractAtomicVar:
    ((∀ i:I (AtomicVar.Interface.PendingReadValue[i] = AbstractAtomicVar.PendingReadValue[i]
     ∧ AtomicVar.Interface.PendingWriteValue[i] = AbstractAtomicVar.PendingWriteValue[i]
     ∧ (AtomicVar.Interface.pc[i] = read_enable ∨ AtomicVar.Interface.pc[i] = read_block
        ∨ AtomicVar.Interface.pc[i] = reading)
             ⇔ AbstractAtomicVar.pc[i] = read_wait
     ∧ (AtomicVar.Interface.pc[i] = read_release ∨ AtomicCar.Interface.pc[i] = read_return)
             ⇔ AbstractAtomicVar.pc = read_done
     ∧ (AtomicVar.Interface.pc[i] = write_enable ∨ AtomicVar.Interface.pc[i] = write_block
        ∨ AtomicVar.Interface.pc[i] = writing)
             ⇔ AbstractAtomicVar.pc[i] = write_wait
     ∧ (AtomicVar.Interface.pc[i] = write_release ∨ AtomicVar.Interface.pc[i] = write_return )
             ⇔ AbstractAtomicVar.pc = write_done
     ∧ AtomicVar.Interface.pc[i] = idle ⇔ AbstractAtomicVar[i] = idle ))
     ∧ AbstractAtomicVar.value = AtomicVar.Interface.value ∧ AbstractAtomicVar.now = AtomicVar.now
     ∧ AbstractAtomicVar.Record ⊆ AtomicVar.Interface.Record
     ∧ ∀ R:OpRec (R ∈ AbstractAtomicVar.Record ∧ (R.op = read_done ∨ R.op = write_done)
                  ⇔ R ∈ AtomicVar.Interface.Record)
     ∧ AbstractAtomicVar.WriteLog = AtomicVar.Interface.WriteLog)
```

Figure 5.24 Simulation relation for the *atomic* objects.

Stated informally these are:

- The pending values for each user process are equivalent.
- The program counter values are related.
- The value of X is equivalent.
- The abstract *Record* log contains all the interval ending entries of the implementation's log.
- The *WriteLog* variables are identical
- Time passes identically.

Table 5.1 lists the equivalent program-counter values for the abstract and concrete automata.

TABLE 5.1 Program Counter Correspondence for *Atomic*

Abstract State	Equivalent States	Abstract State	Equivalent States
read_wait	read_enable read_block reading	write_wait	write_enable write_block writing
read_done	read_release read_return	write_done	write_release write_return
idle	idle	idle	idle

A visual inspection of the start states of both automata is sufficient to indicate that the relations hold initially. We must now check each possible action π of $AtomicVar$.

- Case $\pi = read_i$. The corresponding execution fragment β is $\beta = \{read_i\}$. The only variables affected by the change are the pc_i. Since F maps *read_enable* to *read_wait*, the state correspondence is preserved.
- Case $\pi = read\_value_i(x)$. The corresponding execution fragment β is $\beta = \{read\_value_i(x)\}$. Since we assume that F holds in the initial state, it must also hold in the final state since both pc_i variables are changed to *idle*. No other relevant state changes occur.
- Case $\pi = write_i(v)$. The corresponding execution fragment β is $\beta = \{write_i\}$. This step makes equivalent changes to the $PendingValue_i$ variables. Also, the changes to pc_i preserve the state correspondence since *write_enable* is mapped to *write_wait*.

- Case $\pi = write\_ok_i$. The corresponding execution fragment β is $\beta = \{write\_ok_i\}$. This step sets pc_i to idle in both automata. The correspondence is preserved.
- Case $\pi = try_i$. The corresponding execution fragment β has zero steps. The state relation is unchanged by this transition.
- Case $\pi = crit_i$. The corresponding fragment of β has zero steps. Although the value of $AtomicVar.pc_i$ changes, the mapping from $F(s)$ remains the same.
- Case $\pi = exit_i$. If $AtomicVar.pc_i$ = reading, then $\beta = \{do\_read_i\}$, otherwise $AtomicVar.pc_i$ = writing and $\beta = \{do\_write_i\}$. Thus, if i is currently reading, the value of $PendingValue_i$ will be updated in both automata. If the user is writing, the central value of X, i.e. $value$, is changed to reflect the new value.
- Case $\pi = rem_i$. The corresponding execution fragment, β, has zero steps. Although the value of $s.pc_i$ changes the corresponding value of $u.pc_i$ remains the same. Thus, the state correspondence is preserved.
- Case $\pi = \nu(t)$. The corresponding execution fragment β is $\beta = \{\nu(t)\}$. We permit time to pass identically in the two automata subject to the following constraints. The time between steps for both the abstract and concrete is regulated by the preconditions on $\nu(t)$. Thus, so long as upper and lower time bounds are equal, the effective bounds are the same. If $N = \|I\|$, we can show that the time from an invocation to the execution of an action is bounded by:

$$ReadTime.ub = r\_disp\_bnd.ub + (N-1) \cdot max(r\_exec\_bnd.ub, w\_exec\_bnd.ub) + r\_exec\_bnd.ub \quad (5.13)$$

$$ReadTime.lb = r\_disp\_bnd.lb + r\_exec\_bnd.lb \quad (5.14)$$

$$WriteTime.ub = w\_disp\_bnd.ub + (N-1) \cdot max(r\_exec\_bnd.ub, w\_exec\_bnd.ub) + w\_exec\_bnd.ub \quad (5.15)$$

$$WriteTime.lb = w\_disp\_bnd.lb + w\_exec\_bnd.lb \quad (5.16)$$

The two upper bound equations tracks the time for a process to: dispatch a try_i request, wait for potentially all the other processes to execute an operation, and then perform the operation itself. Similarly, the time from the execution of an action until the external acknowledgement is:

$$ReadResp.lb = r\_return\_bnd.lb \quad (5.17)$$

$$ReadResp.ub = d + r\_return\_bnd.ub \quad (5.18)$$

$$WriteResp.lb = w\_return\_bnd.lb \quad (5.19)$$

$$WriteResp.ub = d + w\_return\_bnd.ub \tag{5.20}$$

That the *AbstractAtomicVar* automaton is permitted to take larger time-steps than the *AtomicVar* automaton is immaterial. The simulation that we are proving, operates in the other direction, i.e. nothing prevents *AbstractAtomicVar* from taking a series of small steps.

Having completely enumerated the possible steps of automaton *AtomicVar*, it is clear that the simulation relation to *AbstractAtomicVar* is valid.

The same procedure can be used to prove that *AtomicVar*2 implements *AbstractAtomicVar*2. The simulation relation G for these automata is given in Figure 5.25. Informally, the meanings of the state correspondences are the same as for the *atomic* case, but the program counter values are more complex (Table 5.2). From the inspection of the automata, the start states satisfy $u \in G(s)$. As before, we must consider each allowable step of automaton *AtomicVar*2 to prove that G is a valid simulation relation.

TABLE 5.2 Program Counter Correspondence for *Atomic2*

Abstract State	Equivalent States	Abstract State	Equivalent States
read_wait	read_enable read_block reading	write_wait	write_enable write_block write_read_enable write_read_block writing
read_done	read_release read_return	write_done	write_read_release notify_enable
idle	idle	notify	notify
		notify_done	notify_release write_release write_done
		idle	idle

- Case $\pi = read_i$. The corresponding execution fragment β is $\beta = \{read_i\}$. The only changed variable that is referenced in the simula-

```
automaton AtomicVar2(v0: DataType, Inf: Real)
    components Interface:AtomicVarInterface(v0, Inf),
               MutEx_r: SemaphoreMutEx(Inf),
               MutEx_w: SemaphoreMutEx(Inf)
           hidden: try_r(i), try_w(i), crit_r(i), crit_w(i), rem_r(i), rem_w(i),
                   exit_r(i), exit_w(i)

invariant I5a of AtomicVar2:
    ∀ i:I ((MutEx_r.Status[i] = crit) ⇔
               (Interface.pc_write[i] = writing ∨ Interface.pc_read[i] = reading))

invariant I5b of AtomicVar2:
    ∀ i:I ((MutEx_w.Statue[i] = crit) ⇔
               (Interface.pc_write[i] ∈
  {write_read_enable, write_read_block, write_read_release, notify_enable,
                                                  notifying, notify_release}

forward simulation from AtomicVar2 to AbstractAtomicVar2:
    ((∀ i:I (AtomicVar2.Interface.PendingReadValue[i] = AbstractAtomicVar2.PendingReadValue[i]
     ∧ AtomicVar2.Interface.PendingWriteValue[i] = AbstractAtomicVar2.PendingWriteValue[i]
     ∧ (AtomicVar2.Interface.pc_read[i] = read_enable
         ∨ AtomicVar2.Interface.pc_read[i] = read_block
         ∨ AtomicVar2.Interface.pc_read[i] = reading)
             ⇔ AbstractAtomicVar2.pc_read[i] = read_wait
     ∧ (AtomicVar2.Interface.pc_read[i] = read_release
         ∨ AtomicVar2.Interface.pc_read[i] = read_return)
             ⇔ AbstractAtomicVar2.pc_read[i] = read_done
     ∧ (AtomicVar2.Interface.pc_write[i] = write_enable
         ∨ AtomicVar2.Interface.pc_write[i] = write_block
         ∨ AtomicVar2.Interface.pc_write[i] = write_read_enable
         ∨ AtomicVar2.Interface.pc_write[i] = write_read_block
         ∨ AtomicVar2.Interface.pc_write[i] = writing)
             ⇔ AbstractAtomicVar2.pc_write[i] = write_wait
     ∧ (AtomicVar2.Interface.pc_write[i] = write_release
         ∨ AtomicVar2.Interface.pc_write[i] = notify_enable )
             ⇔ AbstractAtomicVar2.pc_write[i] = write_done
     ∧ (AtomicVar2.Interface.pc_write[i] = notify)
             ⇔ AbstractAtomicVar2.pc_write[i] = notify
     ∧ (AtomicVar2.Interface.pc_write[i] = notify_release
         ∨ AtomicVar2.Interface.pc_write[i] = write_release
         ∨ AtomicVar2.Interface.pc_write[i] = write_return)
             ⇔ AbstractAtomicVar2.pc_write[i] = notify_done
     ∧ AtomicVar2.Interface.pc[i] = idle ⇔ AbstractAtomicVar2[i] = idle ))
     ∧ AbstractAtomicVar2.value = AtomicVar2.Interface.value
     ∧ AbstractAtomicVar2.now = AtomicVar2.now
     ∧ AbstractAtomicVar2.Record ⊆ AtomicVar2.Interface.Record
     ∧ ∀ R:OpRec ((R ∈ AbstractAtomicVar2.Record
                   ∧ (R.op = read_done ∨ R.op = write_done
                       ∨ R.op = notify_start ∨ R.op = notify_done))
                  ⇔ R ∈ AtomicVar2.Interface.Record))
```

Figure 5.25 Simulation Relation from *AtomicVar2* to *AbstractAtomicVar2*.

tion relation is pc_read_i. Since the new values also correspond, the overall state correspondence holds.

- Case $\pi = write_i(v)$. The execution fragment, β is $\beta = \{write_i(v)\}$. The new values of $PendingWriteValue_i$ are equivalent and the new pc_write_i values obey the mapping G.
- Case $\pi = try\_r_i$. The execution fragment, β, consists of zero steps. Although the program counter advances in the $AtomicVar2$ automata, the corresponding state of $AbstracAtomivVar2$ remains unchanged. Thus the state correspondence is preserved.
- Case $\pi = try\_w_i$. The execution fragment, β, consists of zero steps. Although pc_write_i advances in the implementation automaton, the abstract variable remains unchanged.
- Case $\pi = crit\_r_i$. The execution fragment, β, consists of zero steps. Although the program counter advances in the $AtomicVar2$ automata, the corresponding state of $AbstracAtomivVar2$ remains unchanged and the state correspondence is preserved.
- Case $\pi = crit\_w_i$. The execution fragment, β, consists of zero steps. Although pc_write_i advances in the implementation automaton, the abstract variable remains unchanged. This preserves the state correspondence of G.
- Case $\pi = exit\_r_i$. If $AtomicVar2.pc\_read_i$ = reading, then $\beta = \{do\_read_i\}$, otherwise $AtomicVar2.pc\_write_i$ = writing and $\beta = \{do\_write_i\}$. Both of these options make the appropriate changes to the pending values, $Record$, $WriteLog$, and program counters. This preserves the state correspondence.
- Case $\pi = exit\_w_i$. This action corresponds to an execution fragment β of zero steps. Although $s.pc\_write_i$ advances, the corresponding abstract value is unchanged.
- Case $\pi = rem\_r_i$. The execution fragment, β, consists of zero steps. Although the program counters may advance in the implementation automaton, the abstract variables remains unchanged. This preserves the state correspondence of G.
- Case $\pi = rem\_w_i$. The execution fragment, β, has zero steps. Reasoning is identical to the above case.
- Case $\pi = read\_value_i(x)$. The corresponding execution fragment β is $\beta = \{read\_value_i(x)\}$. Since we assume that G holds in the initial state, it must also hold in the final state since both pc_read_i variables are changed to *idle*. No other relevant state changes occur.
- Case $\pi = notify_i$. The corresponding execution fragment β is given by $\beta = \{notify_i\}$. Both actions make corresponding changes to $Record$ and pc_write_i. Thus G is preserved.

- Case $\pi = write\_ok_i$. The corresponding execution fragment β is $\beta = \{write\_ok_i\}$. This step sets pc_write_i to idle in both automata. The correspondence is preserved.
- Case $\pi = notify\_ack_i$. The corresponding execution fragment β is given by $\beta = \{notify\_ack_i\}$. his step advances pc_write_i both automata. Thus the state correspondence is preserved.
- Case $\pi = \nu(t)$. The corresponding execution fragment β is $\beta = \{\nu(t)\}$. As we saw in the examination of *atomic*. automata, we can relate the time-bounds from the abstract and specialized automata. The bounds for the *read* operations remain the same as those given in Eqns. 5.13, 5.14, 5.17 and 5.18. Because of the more elaborate write protocol, the write bounds must be re-derived. These bounds are derived as follows. We first define the quantity W to be the upper bound on the time from when a process obtains the write-semaphore to when the process releases the semaphore, assuming there is no contention for the read semaphore. Thus:

$$W = (3 \cdot w\_disp\_bnd.ub) + w\_exec\_bnd.ub + d + notify.ub \tag{5.21}$$

Likewise we define the quantity R to be the time required to complete a read operation, assuming no contention:

$$R = r\_exec\_bnd.ub \tag{5.22}$$

Now if there are N processes accessing the object, the longest waiting time that process i will experience is when $N-1$ processes are trying to write at the same time, and i is last in line for the write semaphore. The first process will complete in time W, since all the other processes are waiting on the write-semaphore and cannot contend for the read-semaphore. The next writing process may have to wait an additional time R, to obtain the read semaphore if the first process attempts a read operation in the interim. It can be shown that the worst-case time for the final process to complete the write (i.e. to reach $exit\_r_i$), is:

$$write\_wait.ub = w\_disp\_bnd.ub + (N-1) \cdot W + \frac{(N-1) \cdot (N-2)}{2} R + w\_exec\_bnd.ub \tag{5.23}$$

The lower bound is simply:

$$write\_wait.lb = 2 \cdot w\_disp\_bnd.lb + w\_exec\_bnd.lb \tag{5.24}$$

The completion relations of Eqns. 5.19 and 5.20 are also valid for this automaton[1].

1. Note that the response time is measured from the end of the notify period.

Since the state correspondence G holds for all allowable steps of the automaton $AtomicVar2$, it must represent a valid simulation relation. Thus, we conclude that $traces(AtomicVar2) \subseteq traces(AbstractAtomicVar2)$.

These simulations are useful for several reasons. First, the close examination of the implementation, assures us that the shared variable X is accessed in a safe manner. Second, the abstract atomic objects are easier to compose into the system-level *publish* automata.

5.3.2 System Timer

The *publish* service behaviour is periodic. For each subscriber, *publish* produces a $publish_i(x)$ event, once per subscription period. Since we want to accurately model the implementation of *publish*, the *dispatch* component requires a periodic stimulus to prompt it generate these *publish* messages. The *timer* automaton provides this service to *publish*.

The illustration in Figure 5.12, depicts the external signature of the *timer* automaton. The IOA code for *timer* is provided in Figure 5.26. The design of the signature suggests a distributed interface so that each action, e.g. $tick_i$, goes to a different process P_i. Most of the modelling so far has followed this pattern. In the composition of *publish*, however, we connect a single *dispatch* automaton with a single *timer* automaton. The action subscripts serve more as a tagging mechanism than an enumeration of distinct actions. We retain the subscripts for clarity.

Operations on the *timer* automaton are fairly simple. A timer entry with period, τ, is created using the $TimerSet_i(tag, \tau)$ action. The timer is identified both by the destination process i and an arbitrary integer tag. After receiving acknowledgement that the timer has been created, the remote automaton (*dispatch* in our case), can start the timer with $TimerStart_i(tag)$. Every τ time units, *timer* will generate a $tick_i(tag)$ message. The remote automaton may also cancel a timer with the $TimerStop_i(tag)$ action.

```
uses Conversions
type TimeBound = tuple of lb:Real, ub:Real
type Timer = tuple of i:I, tag: Int, period:Real, active:Bool, start: Real, bnd: TimeBound
type TickRec = tuple of i:I, tag: Int, t: Real

automaton Timer(Inf:Real)
    signature
        input
            timer_set(i:I, tag: Int, period : Real),
            timer_start(i:I, tag: Int),
            timer_stop(i:I tag: Int)
        output
            tick(i:I, tag: Int),
            SetOk(i:I, tag:Int),
            StopOk(i:I, tag:Int),
        time-passage
            ν(t: Real)
    states
        Timers: Set[Timer] := {},
        CanceledTimers : Set[Timers] := {},
        NewTimers : Set[Timers] := {},
        now : Real,
        jitter : Real,
        ack_bnd: TimeBound,   %Constant bound
        record : Array[I,Int,Seq[Real]] := {}
    transitions
        input timer_set(i:I, tag: Int, period : Real)
            eff for T:Timer in Timers do
                    if (T.i = i ∧ T.tag = tag) then
                        Timers := delete(T, Timers)  %Delete any old timer
                    fi
                od
            NewTimers := insert([i,tag, period, false, 0,
                          [now+ ack_bnd.lb, now + ack_bnd.ub]] , NewTimers);
            record[i][tag] := {};
        input timer_start(i:I, tag:Int)
            eff for T:Timer in Timers do
                if  (T.i = i ∧ T.tag = tag) then
                    Timers:= delete(T, Timers);
                    T := set_active(T, true);
                    T := set_start(T, now);
                    T := set_bnd(T, [now + T.period, now + T.period + jitter]);
                    Timers := insert(T, Timers)
                fi
            od
        output SetOk(i:I, tag:Int)
            choose T:Timer where (T ∈ NewTimers)
            pre T ∈ NewTimers ∧ now ≥ T.bnd.lb
            eff NewTimers := delete(T, NewTimers);
            Timers := insert([T.i, T.period, T.active, T.start, [0,Inf]] , Timers)
```

Figure 5.26 IOA specification of the *timer* automaton.

```
output StopOk(i:I, tag:Int)
    choose T:Timer where (T ∈ CanceledTimers)
    pre T ∈ CanceledTimers ∧ now ≥ T.bnd.lb
    eff CanceledTimers := delete(T,CanceledTimers)
input timer_stop(i:I, tag: Int)
    eff for T:Timer in Timers do
        if (T.i = i ∧ T.tag = tag) then
            Timers := delete(T, Timers);
            CanceledTimers := insert([T.i, T.period, false, T.start,
                      [now + ack_bnd.lb, now + ack_bnd.ub]] , CanceledTimers)
        fi
    od
output tick(i:I, tag: Int)
    choose T:Timer where (T ∈ Timers)
    pre (T ∈ Timers ∧ T.i = i ∧ T.tag = tag ∧ T.active ∧ T.bnd.lb ≤ now)
    eff Timers := insert([T.i, T.tag, T.period, T.active, T.start,
                  [T.bnd.lb+ T.period, T.bnd.lb + T.period + jitter]] ,  delete(T,Timers))
    record[i][tag] := record[i][tag] ⊢ now
time-passage ν(t:Real)
    pre ∀ T:Timer ((T ∈ Timers ∨ T ∈ NewTimers ∨ T ∈ CanceledTimers) ⇒
                   ((now + t) ≤ T.bnd.ub))
    eff now := now + t
```

Figure 5.26 IOA specification of the *timer* automaton.

GTA models permit the user to reason about and affect the timing between events through the use of the *first* or *last* variables. Using GTA to directly influence an algorithm is potentially confusing since these variables are a reflection of the partially-synchronous network model, rather than an a direct algorithmic tool. By convention, the *first* and *last* variables are artificial. They are a contrivance that allows sets upper and lower bounds on when an action is enabled. Manipulation of the time-passage action was used to generate subscriber notifications in the abstract specification of *publish*, however we wish to provide a more accurate depiction of the implementation. In our example, *dispatch* requests timing services of the operating system through the *timer* automaton. The internal operation of *timer* is not important, only its interface and external properties. This distinction differentiates between 'real' components and artefacts of the modelling process and improves the clarity of our implementation automata.

The timing related properties of this automaton (Figure 5.27) can be proven with our usual inductive approach. The task is made easier by using an artificial *record* array, that records $tick_i(tag)$ events as they occur.

```
Invariant I1 of Timer:
  ∀ T:Timer ((T ∈ Timers ∧ T.active)⇒
              ((now ≥ T.start + len(record[T.i][T.tag])*T.period)
              ∧ (now ≤ T.start +(len(record[T.i][T.tag])+1)*T.period + jitter)
              ∧ (T.bnd.lb = (len(record[T.i][T.tag])+1)*T.Period + T.Start)
              ∧ (T.bnd.ub = (len(record[T.i][T.tag])+2)*T.Period + T.Start + jitter)))
invariant I2 of Timer:
  ∀ T:Timer (∀ j:Int ((T ∈ Timers ∧ T.active ∧ j≥ 0 ∧ j<len(record[T.i][T.tag]))
     ⇒ ((record[T.i][T.tag][j] - T.start) ≥ (j+1)*T.period)
        ∧ (record[T.i][T.tag][j] - T.start) ≤ ((j+1)*T.period + jitter)))
```

Figure 5.27 Invariants of the *timer* automaton.

Lemma 5.3.1: *In every reachable state of timer, the time variable now is bounded by the currently active timers.*

Proof: Invariant I1 is proven through induction on the states of *timer*. Since initially there are no active timers, the start state clearly satisfies the invariant. Proceeding with the inductive step, we assume that the invariant holds in state s and consider the transition (s, π, s'). We must consider each possible action.

- Case $\pi = TimerSet_i(tag, \tau)$. Since the timer added by this action is not yet active, the invariant is clearly true in s'.
- Case $\pi = TimerStart_i(tag)$. Since this action resets the *Record* and timer structures T, the assertion must hold in s'.
- Case $\pi = TimerStop_i(tag)$. This action will disable an active contract. Thus the invariant must be satisfied in s',
- Case $\pi = tick_i(tag)$. For this action to be enabled $now \geq s.T.bnd.lb = (len(Record[i][T.tag]) + 1) \cdot T.period$. Thus, when the extra entry is added to *Record*, *now* will still satisfy the invariant bounds. Moreover, from the inductive hypothesis, the new values of $s'.T.bnd$ must also satisfy the invariant.
- Case $\pi = \nu(t)$. Since $t > 0$, and from the preconditions to this action, it is clear that *now* cannot advance beyond the invariant bounds. Thus the assertion is satisfied in s'.

- Case $\pi = SetOk_i(tag)$. Since this action does not affect any relevant state variables, the assertion is vacuously satisfied.
- Case $\pi = StopOk_i(tag)$. This acknowledgement does not affect the variables in this invariant. Therefore the invariant is satisfied.

Lemma 5.3.2: *In every reachable state of timer, any active timer will generate a tick message every τ time units, subject to a maximum jitter.*

Proof: We prove invariant I2, through induction on the states of *timer*. Initially, since *record* is empty, the invariant is true. To proceed with the inductive step, we assume that the invariant holds in state s and consider the transition (s, π, s'). Each possible action π, must be considered.

- Case $\pi = TimerSet_i(tag, \tau)$. This action adds a timer to the set of active timers. Since, the action deletes any previously existing timer matching i and tag, the action guarantees that the *active* field of the *TimerRec* tuple is set to false. Thus, the predicate for implication must be false for the new timer, making the implication true. Since any preexisting timers already satisfy the invariant, the assertion must also hold in s'.
- Case $\pi = TimerStart_i(tag)$. Whether or not we assume well-formed interactions (i.e. can the user start a timer already started?), this action activates the timer T and sets up the time-bounds for it to fire. Since the action clears $record[i][tag]$, and resets the start time, the assertion is clearly satisfied in s'.
- Case $\pi = tick_i(tag)$. If this action is enabled in s, then $now \geq s.T.bnd.lb$. It follows from the invariant hypothesis that if $j = len(Record[i][tag])$ then $s.T.bnd.lb = (s.j + 2) \cdot s.T.period$. Thus, it is clear that as we add another element to *record*, the assertion will continue to hold.
- Case $SetOk_i(tag)$. This action vacuously satisfies the invariant.
- Case $StopOk_i(tag)$. This action vacuously satisfies the invariant.

Theorem 5.3.1: *All active timers in the timer automaton will generate periodic tick actions with uncertainty timing uncertainty, jitter.*

Proof: Since time-passage is restricted by the active timers (Lemma 5.3.1), and historic accuracy is guaranteed (Lemma 5.3.2), the *timer* automaton must provide a *tick* within a narrow window of time width *jitter*.□

5.3.3 The *publish* Dispatcher

The *dispatch* automaton is the last element in the composition of *publish*. Presenting a distributed interface to the external user processes *dispatch* is responsible for setting up subscriptions and dispatching copies of the shared variable X at appropriate intervals. The IOA code for the automaton is shown in Figure 5.28.

There are no invariants to prove for *dispatch*. It's operation is fairly straightforward, and serves mainly to coordinate the other automata in the composition of *publish*. When a user sends a subscribe request, the *dispatch* automaton creates a timer, and then starts it. The periodic $tick_i(tag)$ messages, trigger *dispatch* to read the current *atomic* value of X and send it on to process i.

5.3.4 The *publish_on_change* Dispatcher

Finally, we turn our attention to the *dispatch2* automaton. This automaton completes the composition of the *publish_on_change* service. In many respects this automaton is simpler than *dispatch* since its operation is reactive. It records, the beginning of a subscription, but takes no action until a user writes to the atomic object. When *dispatch2* receives a $notify_i$ message, it generates publish messages for all subscribed processes. The IOA code for this automata is shown in Figure 5.29.

Like the previous example, *dispatch2* has no invariants. It's job is to manipulate the operations of the other components of *publish_on_change*. In the next section we show that the composition implements the abstract service.

5.4 Composition of Publish-Subscribe Services

5.4.1 The *publish* Composition

In Section 5.2.2, we presented the abstract specification if the *publish* service. This generic automaton presented a clear description of the performance of the GRRDE ser-

```
type cmd = enumeration of Sub, Cancel, Set, Stop, SubOk, Start, nil, SetOk, StopOk
automaton Dispatch(v0:Int)
    signature
        input
            subscribe (i:I, T:Real),
            cancel(i:I),
            read_value(i:I, x:X),
            tick(i:I, tag:Int),
            set_ok(i:I, tag:Int),
            stop_ok(i:I, tag:Int)
        output
            Publish(i:I, T:TxRec),
            sub_ok(i:I),
            cancel_ok(i:I),
            timer_set(i:I, tag:Int, period: Real),
            timer_stop(i:I, tag:Int),
            timer_start(i:I, tag:Int),
            read(i:I)
        time-passage
            ν(t:Real)
    states
        PendingCommand:Array[I,cmd] := constant(nil),
        PendingDispatch:Array[I,Seq[X]] := constant({}),
        ReadPending: Array[I,Int] := constant(0),
        TimerTag: Array[I,Int] := constant(0),
        Period: Array[I,Real] :=  constant(0),
        now:Real := 0,
        subscribed: Array[I,bool] := constant(false),
        StepBound:Array[I,TimeBound] := constant([0,Inf]),
        DispatchBound: Array[I,TimeBound] := constant([0,Inf]),
        StartTime:Array[I,Real] := constant(0),
        xmit:TimeBound := [lb,ub]
    transitions
        input subscribe(i:I, T:Real)
            eff Period[i] := T;
            PendingCommand[i] := Set;
            Subscribed[i]:= true;
            StepBound[i] := [now,now];
            TimerTag[i] := 1
        input cancel(i:I)
            eff Period[i] := Inf;
            PendingCommand := Stop;
            subscribed[i] := false;
            StepBound[i] := [now,now]
        input read_value(i:I, x:X)
            eff PendingDispatch[i] := PendingDispatch[i] ⊢ x;
            PendingCommand[i] := nil;
            DispatchBound[i] := [now,now]
        input tick(i:I, tag:Int)
```

Figure 5.28 The *dispatch* automaton model.

```
    eff if subscribed[i] then
        ReadPending[i] := ReadPending[i] + 1;
        DispatchBound[i] := [now,now]
    fi
input set_ok(i:I, tag:Int)
    eff PendingCommand[i] := Start;
    StepBound[i] := [now, now]
input stop_ok(i:I, tag:Int)
    eff PendingCommand[i] := Cancel;
    StepBound[i] := [now + xmit.lb, now + xmit.ub]
output Publish(i:I, T:TxRec)
    pre (len(PendingDispatch[i]) ≠ 0
         ∧ T = [head(PendingDispatch[i]),[now + xmit.lb, now + xmit.ub]])
         ∧ now ≥ DispatchBound[i].lb
    eff PendingDispatch[i] := tail(PendingDispatch[i]);
    DispatchBound[i] := [0,Inf]
output sub_ok(i:I)
    pre PendingCommand[i] = SubOk ∧ now ≥ StepBound[i].lb
    eff PendingCommand[i] := nil;
    StepBound[i] := [0,Inf]
output cancel_ok(i:I)
    pre PendingCommand[i] = Cancel ∧ now ≥ StepBound[i].lb
    eff PendingCommand[i] := nil;
    StepBound[i] := [0,Inf]
output timer_set(i:I, tag:Int, period:Real)
    pre TimerTag[i] = 0 ∧ tag = 1 ∧ period = Period[i]
                    ∧ PendingCommand[i] = Set ∧ now ≥ StepBound[i]
    eff TimerTag[i] := tag;
    PendingCommand[i] := SetOk;
    StepBound[i] := [0,Inf];
output timer_stop(i:I, tag:Int)
    pre tag = TimerTag[i] ∧ PendingCommand[i] = stop ∧ now ≥ StepBound[i]
    eff PendingCommand := StopOk;
    StepBound[i] := [0,Inf]
output timer_start(i:I, tag:Int)
    pre PendingCommand[i] = Start ∧ tag = TimerTag[i] ∧ now ≥ StepBound[i]
    eff StepBound[i] := [now + xmit.lb, now + xmit.ub];
    PendingCommand[i] := SubOk;
    StepBound[i] := [0,Inf];
    StartTime[i] := now
output read(i:I)
    pre ReadPending[i] > 0 ∧ now ≥ DispatchBound[i].lb
    eff ReadPending[i] := ReadPending[i] - 1;
    DispatchBound[i] := [0, Inf]
time-passage ν(t:Real)
    pre (∀ i:I (((now + t) ≤ DispatchBound[i].ub) ∧ (now + t) ≤ StepBound[i].ub))
    eff now := now + t
```

Figure 5.28 The *dispatch* automaton model.

vice, but said little about how those functions were achieved. We have subsequently revealed that the implementation of *publish* is accomplished by the composition of an

```
type cmd = enumeration of Sub, Cancel, Set, Stop, SubOk, Start, nil, SetOk, StopOk
type PCStep = enumeration of idle, read, wait,  ack

automaton Dispatch(v0:V)
    signature
        input
            subscribe (i:I),
            cancel(i:I),
            read_value(i:I, x:X),
            notify(i:I)
        output
            Publish(i:I, T:TxRec),
            sub_ok(i:I),
            cancel_ok(i:I),
            notify_ack(i:I),
            read(i:I)
        time-passage
            ν(t:Real)
    states
        dispatching:I := NULL,
        dispatchPC : PCStep := idle,
        PendingCommand:Array[I,cmd] := constant(nil),
        Dispatch_Seq:Array[I,Seq[TxRec]] := constant({}),
        now:Real := 0,
        subscribed: Array[I,bool] := constant(false),
        StepBound:Array[I,TimeBound] := constant([0,Inf]),
        ReadBound: TimeBound := [0,Inf],
        dispatch: TimeBound := [0,1],
        StartTime:Array[I,Real] := constant(0),
        xmit:TimeBound := [lb,ub]
    transitions
        input subscribe(i:I)
            eff Period[i] := T;
            PendingCommand[i] := SubOk;
            Subscribed[i]:= true;
            StepBound[i] := [now + xmit.lb, now + xmit.ub]
        input cancel(i:I)
            eff PendingCommand := Cancel;
            subscribed[i] := false;
            StepBound[i] := [now + xmit.lb, now + xmit.ub]
        input read_value(i:I, x:X)
            eff for j:I so that subscribed[j] do
                DispatchSeq[j] := DispatchSeq[j] ⊢ [x,[now + dispatch.lb, now + dispatch.ub]]
            od;
            ReadBound := [now, now];
            dispatchPC := ack;
            dispatching := NULL;
        output Publish(i:I, T:TxRec)
            pre (len(DispatchSeq[i]) ≠ 0
                 ∧ T = [head(DispatchSeq[i]),[now + xmit.lb, now + xmit.ub]] )
```

Figure 5.29 IOA model of *dispatch2* automaton.

```
        ∧ now ≥ head(DispatchSeq[i]).lb
    eff PendingDispatch[i] := tail(PendingDispatch[i]);
output sub_ok(i:I)
    pre PendingCommand[i] = SubOk ∧ now ≥ StepBound[i].lb
    eff PendingCommand[i] := nil;
    StepBound[i] := [0,Inf]
output cancel_ok(i:I)
    pre PendingCommand[i] = Cancel ∧ now ≥ StepBound[i].lb
    eff PendingCommand[i] := nil;
    StepBound[i] := [0,Inf]
input notify(i:I)
    eff dispatching := i;
    dispatchPC := read;
    ReadBound := [now, now]
output read(i:I)
    pre dispatching = i ∧ dispatchPC = read ∧ now ≥ ReadBound[i].lb
    eff dispatchPC := wait;
    ReadBound[i] := [0, Inf]
output notify_ack(i:I)
    pre i = dispatching ∧ dispatchPC = ack ∧ now ≥ ReadBound.lb
    eff dispatchPC := idle;
    ReadBound := [0,inf]
time-passage ν(t:Real)
    pre ( (now + t) < ReadBound.ub ∧ (∀ i:I ((now + t) ≤ StepBound[i].ub)))
    eff now := now + t
```

Figure 5.29 IOA model of *dispatch2* automaton.

AbstractAtomicVar, a *timer*, and a *dispatch* automaton. Although, we have detailed the construction of these components, we must also show that their combined operation implements the abstract publish service. Consider the proposed simulation relation, F, shown in Figure 5.30. This relation contends that the log variables of *publish* are duplicated in a distributed fashion across the composed automaton. Note that the composition has been named *ComposedPub*. To show that $u \in F(s)$, we must consider both the initial conditions, and the step correspondence. From inspection of the automata, we can conclude that F holds in the start state. Each action π of *ComposedPub* must be considered individually.

- Case $\pi = subscribe_i(\tau)$. The corresponding execution fragment β is given by $\beta = \{subscribe_i(\tau)\}$. This action makes corresponding changes to the $period_i$, $subscribed_i$ and $PendingCommand_i$ variables in both automata. Thus, the state correspondence is preserved.
- Case $\pi = cancel_i$. $\beta = \{cancel_i\}$. Equivalent changes are made to both automata. Thus, the state correspondence is preserved.

```
automaton ComposedPub:
    components dispatch:dispatch,
               atomic:AbstractAtomicVar,
               timer:timer
    hidden  read, read_value, TimerSet, TimerStop,
            SetOk, StopOk, TimerStart, Tick

forward simulation from ComposedPub to publish:
   (publish.now = ComposedPub.now
    ∧ publish.WriteLog = ComposedPub.atomic.WriteLog
    ∧ publish.pub_value = ComposedPub.atomic.value
    ∧ ∀ i:I ((publish.Subscribed[i] = ComposedPub.dispatch.subscribed[i])
               ∧ (publish.Period[i] = ComposedPub.dispatch.Period[i])
               ∧ (publish.PendingCommand[i] = cancel
                    ⇔ (ComposedPub.dispatch.PendingCommand[i] = cancel
                         ∨ ComposedPub.dispatch.PendingCommand[i] = Stop
                         ∨ ComposedPub.dispatch.PendingCommand[i] = StopOk))
               ∧ (publish.PendingCommand[i] = sub
                    ⇔ (ComposedPub.dispatch.PendingCommand[i] = set
                         ∨ ComposedPub.dispatch.PendingCommand[i] = start
                         ∨ ComposedPub.dispatch.PendingCommand[i] = SubOk))
               ∧ (publish.PendingCommand[i] = Commit
                    ⇔ ComposedPub.atomic.pc[i] = write_wait)
               ∧ (publish.StartTime[i] = ComposedPub.dispatch.StartTime[i])))
```

Figure 5.30 Simulation relation from *ComposedPub* to *Publish*.

- Case $\pi = read\_value_i(x)$. $\beta = \{\varnothing\}$. Since the action is not externally observable, and does not modify any relevant state variables, the correspondence must hold.
- Case $\pi = tick_i(tag)$. $\beta = \{\varnothing\}$. Since the action is not externally observable, and does not modify any relevant state variables, the correspondence must hold.
- Case $\pi = SetOk_i(tag)$. $\beta = \{\varnothing\}$. Even though the $PendingCommand_i$ variable changes in *ComposedPub*, the corresponding value from $F(s)$ remains the same. Thus, the state correspondence is maintained.
- Case $\pi = StopOk_i(tag)$. $\beta = \{\varnothing\}$. Even though the $PendingCommand_i$ variable changes in *ComposedPub*, the corresponding value from $F(s)$ remains the same. Thus, the state correspondence is maintained.
- Case $\pi = publish_i(T)$. $\beta = \{publish_i(T')\}$. Since *dispatch* does not allow the progress of time between receiving a read_value$_i(x)$ and generating the corresponding $publish_i(T)$ event, and from *AbstractAtomicVar*, a non-zero amount of time must pass between a read-return and a subsequent write, therefore it must be the case that the published value corresponds to *ComposedPub.atomic.value*. Hence, $T = T'$, the external traces match, and the state correspondence is preserved.

- Case $\pi = sub\_ok_i$. $\beta = \{sub\_ok_i\}$. Both automata make appropriate changes to their state variables, preserving the state correspondence.
- Case $\pi = cancel\_ok_i$. $\beta = \{cancel\_ok_i\}$. As in the above case, the changes made to both automata are complementary and the state correspondence is preserved.
- Case $\pi = TimerSet_i(tag)$. $\beta = \{\varnothing\}$. Even though the $PendingCommand_i$ variable changes in $ComposedPub$, the corresponding value from $F(s)$ remains the same. Thus, the state correspondence is maintained.
- Case $\pi = TimerStop_i(tag)$. $\beta = \{\varnothing\}$. Even though the $PendingCommand_i$ variable changes in $ComposedPub$, the corresponding value from $F(s)$ remains the same. Thus, the state correspondence is maintained.
- Case $\pi = TimerStart_i(tag)$. $\beta = \{\varnothing\}$. This action alters the $PendingCommand_i$ for the composed automata, but the corresponding value in $Publish$ unchanged and F holds.
- Case $\pi = read_i$. $\beta = \{\varnothing\}$. This action does not affect any state variables mentioned in the simulation relation. Thus, the correspondence is trivially satisfied.
- Case $\pi = write_i(v)$. $\beta = \{write_i(v)\}$. This action changes $publish.pending_i$ and $ComposedPub.atomic.PendingValue_i$ in the same manner, thus preserving the state correspondence.
- Case $\pi = do\_read_i$. $\beta = \{\varnothing\}$. This action does not affect the state variables mentioned in F.
- Case $\pi = do\_write_i$. $\beta = \{WriteCommit_i\}$. This action sets the stored value of X from the pending write value. This pending value must be the same for both abstract and implementation automata. Thus, the state correspondence is preserved.
- Case $\pi = write\_ok_i$. $\beta = \{write\_ok_i\}$. Since this action makes compatible changes to $PendingCommand$ in both automata, the state correspondence is preserved.
- Case $\pi = \nu(t)$. $\beta = \{\nu(t)\}$. If the time-passage action in both automata are enabled for a certain value of t, it is clear that the state correspondence must be maintained. If we are to prove that both actions are enabled, we must relate the upper bounds on time-passage between the $publish$ and $ComposedPub$ automata. As long as the upper bounds for the abstract automaton are at least as large as that for the concrete, the simulation relation is valid. Rather, than *prove* state correspondence for these variables, we derive relations between them:

$$\begin{aligned} SubOk.ub &= 2 \cdot AckBnd.ub + xmit.ub \\ SubOk.lb &= 2 \cdot AckBnd.ul + xmit.lb \end{aligned} \tag{5.25}$$

$$\begin{aligned} CancelOk.ub &= AckBnd.ub + xmit.ub \\ CancelOk.lb &= AckBnd.lb + xmit.lb \end{aligned} \tag{5.26}$$

$$pub\_jitter = timer.jitter + read\_time.ub + read\_resp.ub \tag{5.27}$$

$$\begin{aligned} WriteCommitTime.ub &= write\_time.ub \\ WriteCommitTime.lb &= write\_time.lb \end{aligned} \tag{5.28}$$

$$\begin{aligned} WriteOk.ub &= write\_ok.ub \\ WriteOk.lb &= write\_ok.lb \end{aligned} \tag{5.29}$$

Therefore F is a valid simulation from $ComposedPub$ to $publish$ and $traces(ComposedPub) \subseteq traces(publish)$.

The derived timing relations can be used to derive the worst case time bounds of the GRRDE $publish$ service. Since $cancel_i$ and $subscribe_i$ actions are primarily transient activities, they are of minor importance in steady state operations. The most important metrics in evaluating system performance are pub_jitter and $WriteCommitTime.ub$. From Eqns 5.13, 5.15, 5.28, and 5.29:

$$\begin{aligned} WriteCommitTime.ub = {} & w\_disp\_bnd.ub + w\_exec\_bnd.ub \\ & + (N-1) \cdot max(r\_exec\_bnd.ub, w\_exec\_bnd.ub) \end{aligned} \tag{5.30}$$

and,

$$\begin{aligned} pub\_jitter = {} & timer.jitter + r\_disp\_bnd.ub + r\_exec\_bnd.ub + d + r\_return\_bnd.ub \\ & + (N-1) \cdot max(r\_exec\_bnd.ub, w\_exec\_bnd.ub) \end{aligned} \tag{5.31}$$

We expect that the bound on the process jitter will scale with the number of potential subscribers. The full impact of these relations on the design of real-time systems with GRRDE will be assessed in Chapter 7.

5.4.2 The *publish_on_change* Composition

We conclude the formal derivations and proofs by showing the simulation G relation expresses the state correspondence between the implementation of *publish_on_change* and its abstract specification. To form the composition, $ComposedPub2$, we combine one $dispatch2$ automaton and one $atomic2$ automaton. As before, the relation, G, relates the states, s, of $ComposedPub2$, to the states, u, of *publish_on_change*. The simulation relation is given in Figure 5.31.

```
automaton ComposedPub2
    components dispatch:dispatch2
               atomic: AbstractAtomicVar2
    hidden notify, notify_ack, read, read_value
forward simulation from ComposedPub2 to publish_on_change:
   (publish_on_change.now = ComposedPub2.now
    ∧ publish_on_change.WriteLog = ComposedPub2.atomic.WriteLog
    ∧ publish_on_change.pub_value = ComposedPub2.atomic.value
    ∧ publish_on_change.DispatchSeq = ComposedPub2.dispatch.DispatchSeq
    ∧ ∀ i:I ((publish_on_change.Subscribed[i] = ComposedPub2.dispatch.subscribed[i])
              ∧ (publish_on_change.Period[i] = ComposedPub2.dispatch.Period[i])
              ∧ (publish_on_change.PendingCommand[i] = cancel
                   ⇔ (ComposedPu2b.dispatch.PendingCommand[i] = cancel
                        ∨ ComposedPub2.dispatch.PendingCommand[i] = Stop
                        ∨ ComposedPub2.dispatch.PendingCommand[i] = StopOk))
              ∧ (publish_on_change.PendingCommand[i] = sub
                   ⇔ (ComposedPub2.dispatch.PendingCommand[i] = set
                        ∨ ComposedPub2.dispatch.PendingCommand[i] = start
                        ∨ ComposedPub2.dispatch.PendingCommand[i] = SubOk))
              ∧ (publish_on_change.PendingCommand[i] = Commit
                   ⇔ ComposedPub2.atomic.write_pc[i] = write_wait)
              ∧ (publish_on_change.StartTime[i] = ComposedPub.dispatch.StartTime[i])))
```

Figure 5.31 Simulation relation from *ComposedPub2* to *publish_on_change*.

In the start states of both automata, the relationship is clearly valid. The remaining task is to examine each step of $ComposedPub2$ and provide a step correspondence in *publish_on_change*.

- Case $\pi = subscribe_i$. The corresponding execution fragment β is given by $\beta = \{subscribe_i\}$. This step makes identical changes to the $subscribed_i$ and $StartTime_i$ variables of both automata.

- Case $\pi = cancel_i$. $\beta = \{cancel_i\}$. Equivalent changes to the $subscribed_i$ and $PendingCommand_i$ variables are made in both automata. Thus, the state correspondence mandated by G is preserved.
- Case $\pi = read\_value_i(x)$. $\beta = \{GenerateMessages_i\}$. This step adds the appropriate $TxRec$ messages to the $DispatchQueue_i$ variables in $ComposedPub2$ and *publish_on_change*.
- Case $\pi = publish_i(T)$. $\beta = \{publish_i(T')\}$. Since $dispatch2$ does not allow the progress of time between receiving a read_value$_i(x)$ and generating the corresponding $publish_i(T)$ event, and $AbstractAtomicVar2$ requires a non-zero amount of time to pass between a read-return and a subsequent write, therefore it must be the case that the published value corresponds to $ComposedPub2.atomic.value$. Hence, $T = T'$, the external traces match, and the state correspondence is preserved.
- Case $\pi = sub\_ok_i$. $\beta = \{sub\_ok_i\}$. Only the value of $PendingCommand_i$ is altered. Since $u \in G(s)$ and $u' \in G(s')$, the state correspondence is maintained.
- Case $\pi = cancel\_ok_i$. $\beta = \{cancel\_ok_i\}$. This action has identical effects in both automata, thus preserving G.
- Case $\pi = read_i$. $\beta = \{\emptyset\}$. Since, $read_i$ has no external trace, and no state variables referenced in G are altered, the state correspondence is preserved.
- Case $\pi = write_i(v)$. $\beta = \{write_i(v)\}$. This action makes compatible changes to both the $PendingWriteValue_i$ and $PendingCommand_i$ variables. We observe that the state correspondence is maintained.
- Case $\pi = do\_read_i$. $\beta = \{\emptyset\}$. This action has no external trace and does not affect any relevant variables. Thus, the state correspondence is trivially satisfied.
- Case $\pi = do\_write_i$. $\beta = \{WriteCommit_i\}$. Since the pending write values in both automata are the same, the value written to X and to $WriteLog$ must also be the same. This preserves the state correspondence.
- Case $\pi = notify_i$. $\beta = \{\emptyset\}$. This step has no externally visible trace, and does not alter the referenced state variables in $ComposedPub2$.
- Case $\pi = notify\_ack_i$. $\beta = \{\emptyset\}$. This step also has no externally visible trace, and does not alter the referenced state variables in $ComposedPub2$.
- Case $\pi = write\_ok_i$. $\beta = \{write\_ok_i\}$. The external traces and variable changes made by this action clearly preserve the state correspondence.
- Case $\pi = \nu(t)$. $\beta = \{\nu(t)\}$. As discussed in the simulation proof for *publish*, the effect of this transition clearly satisfies state correspondence. The remaining task is to show that both actions are enabled. Since the imple-

mentation automaton frequently makes several internal steps for each step of the abstract automaton, we again specify a least upper bound on the bounding constraints of *publish_on_change*.

$$\begin{aligned} SubOk.ub &= DispStep.ub + xmit.ub \\ SubOk.lb &= DispStep.ul + xmit.lb \end{aligned} \tag{5.32}$$

$$\begin{aligned} CancelOk.ub &= DispBnd.ub + xmit.ub \\ CancelOk.lb &= DispBnd.lb + xmit.lb \end{aligned} \tag{5.33}$$

$$\begin{aligned} WriteCommitTime.ub &= write\_time.ub \\ WriteCommitTime.lb &= write\_time.lb \end{aligned} \tag{5.34}$$

$$\begin{aligned} generate\_bound.ub &= notify\_time.ub + read\_time.ub \\ generate\_bound.lb &= notify\_time.lb + read\_time.lb \end{aligned} \tag{5.35}$$

$$\begin{aligned} WriteOk.ub &= write\_ok.ub \\ WriteOk.lb &= write\_ok.lb \end{aligned} \tag{5.36}$$

$$publish\_on\_change.dispatch\_bound = ComposedPub2.dispatch.dispatch\_bound \tag{5.37}$$

Since each step of the automaton *ComposedPub2* preserves the state correspondence G, the simulation relation is valid. Therefore we can say that *ComposedPub2* implements *publish_on_change* and $traces(ComposedPub2) \subseteq traces(publish\_on\_change)$.

The worst case timing bounds can be developed further. The critical operations are: the maximum time between a write invocation and a commit, $WriteCommitTime.ub$, and the maximum delay between a committed write and the corresponding *Publish* events, Δ_{pub}. From the above formulae and Eqns. 5.21, 5.22 and 5.23 we can show that:

$$WriteCommitTime.ub = w\_disp\_bnd.ub + (N-1)\cdot W + \frac{(N-1)\cdot(N-2)}{2}R + w\_exec\_bnd.ub \tag{5.38}$$

At first glance, that this bound will grow dangerously large due to the N^2 dependency. Further reflection on the real system operation suggests that this figure is rather conservative. Good GRRDE designs restrict the number of processes with *write* access to the published variable to N_w. Rarely would there be more than a few processes within a software

module writing to a published value. If we separately account for the number of possible subscribers, N_s we get:

$$WriteCommitTime.ub = w\_disp\_bnd.ub + (N_w - 1) \cdot (W + N_s \cdot R) + w\_exec\_bnd.ub \tag{5.39}$$

As the system grows, the maximum commit time will grow linearly with the number of subscribers in the system. Since all processes within a destination module will likely share a subscription, N_s refers to the number of functional blocks, rather than the number of system processes.

The publish delay can also be examined more carefully.

$$\Delta_{pub} = generate\_bound.ub + dispatch\_bound.ub \tag{5.40}$$

which becomes

$$\begin{aligned} \Delta_{pub} = {} & r\_disp\_bnd.ub + (N_s - 1) \cdot r\_exec\_bnd.ub \\ & + r\_exec\_bnd.ub + notify\_time.ub + dispatch\_bound.ub \end{aligned} \tag{5.41}$$

This quantity also grows linearly with the number of subscribers in the system. This growth is to be expected since, the dispatch functions must be executed for each subscriber.

5.5 Summary

In this chapter we have applied formal analysis techniques to the GRRDE publish-subscribe services. Presentation of the abstract specification provides an unambiguous statement of the properties of the middleware system. We subsequently provided detailed general timed automata models of the actual algorithms employed in the run-time system. Using simulation relations, we were able to prove that these software algorithms satisfactorily assert the same timing and correctness properties of the abstract specifications. In the process of establishing these simulations, we derived detailed timing bounds for the various publish-related operations.

Although our formal verification examines the GRRDE algorithms and not the actual GRRDE source code, this approach provides the most cost effective use of time and resources. In the next chapter, we examine on-line testing of actual GRRDE applications. This testing was used to derive estimates of the timing parameters derived above and to improve our confidence that the our algorithms were correctly implemented.

Chapter 6

TESTING AND CHARACTERIZATION

Software engineers in the past have hoped that formal analysis alone would be enough to assure system correctness [Tanenbaum, 1976]. Realistically, formal methods are just one tool in the validation process. For real-time systems in particular, temporal characterizations of the host system are important. Formal analysis may provide parameterized bounds on system behaviour, but before the relations can be useful, these quantities must be measured.

This chapter presents the testing and characterization of the GRRDE publish-subscribe services. Our focus is not so much in the removal of errors[1], but in building greater understanding of how the system behaves. The formal derivations in the previous chapter gave us an understanding of the qualitative nature of the temporal performance bounds. Here, we seek to provide direct measurements that can be used to support systems design. It is inevitable that generic middleware such as GRRDE will incur some performance overhead. These results provide perspective on the 'cost' of GRRDE-based design.

GRRDE has been examined from both a static and dynamic standpoint. We begin with a discussion of GRRDE memory requirements. This is followed by some basic timing measurements for the PowerPC embedded computers and the OSE operating system. We then present the more elaborate test-suites developed to measure performance of the publish-

1. We conducted separate debugging tests, but they are not relevant to this discussion.

subscribe services. We conclude with comparisons to published tests for other middleware systems.

6.1 Memory Requirements

Memory is often taken for granted these days. It is not unusual for PCs to have more system memory than the hard drive space of seven years ago. Dynamic memory management allows computers to respond to a user's widely varying usage patterns. In embedded systems, memory usage is much more circumspect. Many real-time operating systems do not even support dynamic memory allocation. In contrast to the general purpose PC, embedded devices are usually employed for a much narrower series of tasks. This permits the developer to specify absolute, *a priori* bounds on the memory usage for all system functions.

This section examines the memory overhead involved in using the GRRDE services. We first present a quick overview of how memory is organized under OSE. With this understanding, we can analyze and predict the memory usage of an evolving simulation.

6.1.1 The OSE Memory Model

The OSE real-time operating system (RTOS) permits flexible and powerful memory management. Memory can be used for three primary purposes. The first use of memory is *code* memory. These regions store the actual machine instructions that a process runs when it executes. The machine instructions for all executable processes must be stored somewhere in memory before the process can be created. The second variety is not truly memory, but *memory-mapped I/O*. This useful abstraction makes the interface to all input and output devices appear to be special places in memory. To control the device, the user writes to the particular locations. Lastly, *data* memory complements the code memory and makes up the bulk of the remaining space. This storage is used to hold all the variables in the system.

The notion of data memory is rather ambiguous. This memory is actually used in many different ways. Not only do OSE blocks group processes together, but they also define protected segments of memory. Each block contains two areas of memory: *static data* and the *pool*. The static data area contains allocations for all global and file-scope variables. It is created implicitly when the system is compiled. The pool is dimensioned by the user in the system configuration files.

Pool usage can be decomposed further. OSE permits a limited form of dynamic memory allocation using *stack buffers* and *signal buffers*. Each process is allocated a stack buffer upon creation. This buffer is used to allocate process- and function-scope variables. Stack space is also used to store context information (e.g. registers, the program counter, etc.) that must be maintained when the process is preempted. Signal buffers are used for inter-process communication. Both types of buffers are allocated from the block's pool when necessary and must be one of a number of fixed sizes. Buffers can be recycled (e.g. when the message arrives, or the process is killed), but they cannot be returned to the pool or resized.

The same basic memory allocation scheme is used, regardless of whether the user's applications are compiled into a monolithic kernel, or a loadable module. Users must be aware that extensive use of loadable modules creates memory inefficiencies. Although the operating system provides hooks and 'stub' functions for system functions, the machine instructions for most library calls (e.g. math functions, I/O routines, etc.) are duplicated in each module. This includes duplication of the code for the GRRDE services. Monolithic kernels avoid this inefficiency by including only one copy of the machine instructions.

6.1.2 OSE Memory Usage

The software that implement the publish-subscribe services in GRRDE are inherently distributed. Each simulation module contains several processes that implement certain GRRDE related functions, and specialized access routines must be included when compiling the user's applications. Consequently, each block in the system must include a little

extra memory to support the use of GRRDE. Memory allocation for internal processes and data-structures is statically allocated when a module is compiled. Memory use can be split into three categories:

Code Memory

This memory contains the machine language instructions for each element of the GRRDE package. Key components include: byte-order conversion routines (*big_little*), key processes and contract classes (*gflops_base*), signal, contract, and process wrapper classes (*gflops_wrap*), time manipulation routines (*sim_time*), and process identification routines (*proc_locate*). This memory usage is duplicated for each loadable module but is only included once in a monolithic system.

Static Data

The subscription handling routines maintain two internal tables in each simulation module. These tables are statically sized but can be adjusted in the source code to optimize memory usage for a particular system configuration. The default sizes of tables include storage for fifty simultaneous subscriptions (184 bytes each), and fifty registered services (48 bytes each). This storage is needed for each software block.

Pool Data

Each GRRDE process requires a stack buffer. Since most of the shared data exists at the file scope, these processes need little stack space. This storage is needed in each simulation block.

The memory utilization for each of the above categories was measured by examining the configuration and object files. A summary of the results is shown in Table 6.1.

From these results we can estimate the total memory overhead M_{tot} (in KB), for a system of N simulation modules.

For a CPU using only loadable modules we have:

TABLE 6.1 GRRDE Memory Usage

Component	Size (Bytes)
big_little	124
gflops_wrap	5206
gflops_base	12396
sim_time	2140
proc_locate	868
Code Memory	**20636**
Contract Table	9200
Dispatch Table	960
Static Data	**10160**
block_manager	2000
input_arbiter	1000
message_negotiator	1000
message_dispatcher	1000
Pool Memory	**5000**
Total	**35796**

$$M_{tot_{mod}} = 35 \cdot N \tag{6.1}$$

and for a monolithic kernels this becomes:

$$M_{tot_{mono}} = 25 \cdot N + 10 \tag{6.2}$$

It should be noted that we did not perform any memory optimizations when compiling the source-code for these measurements. It is reasonable to assume that these figures are conservative upper bounds on the requirements of mature simulations.

Spacecraft systems vary in the amount of memory available to them. Mars Pathfinder had quite a large amount of memory (16 MB) [Chau, et al, 1995]. For missions such as this, several hundred kilobytes could probably be allocated for a system like GRRDE without significantly altering the memory budget. Smaller missions, such as the Stanford University OPAL [Twiggs, et al, 1999] mission have more modest memory capacities (1MB).

These systems might be able to accommodate a moderately optimized version of GRRDE. Satellites like PACSAT [Diersing, 1993] without much memory (256K), would require a much smaller memory footprint. It is unlikely that GRRDE would be viable for these missions. These comparisons should not be taken as discouraging since GRRDE is aimed at supporting *complex, processing intensive* missions. Missions attempting to make use of advanced software capabilities are precisely those that are likely to have the memory budget to support GRRDE.

6.2 General Timing Benchmarks

Before discussing the specific timing results of the publish subscribe services, it is insightful to consider the general temporal behaviour of the GFLOPS embedded processors. We present measured arithmetic benchmarks, and published OSE measurements.

6.2.1 PowerPC Arithmetic Benchmarks

Benchmarking is an essential part of embedded systems development [Stewart, 1999]. Fine-grain, hand optimization, should not be done prematurely, but a clear understanding of processing inefficiencies helps the developer prioritize improvements. Table 6.2 shows the results of a series of arithmetic tests made using OSE. Instruction and data caches were disabled for these tests along with all compiler optimizations.

Floating point operations on this processor are very fast. Fixed point arithmetic may also be more efficient than observed during testing since our test structure nullifies the benefits of the processor cache and secondary arithmetic logic unit [Motorola, 2001]. Important bottlenecks to note are the very high cost of floating-point division, and fixed-to-floating point type promotion. The timing measurements for these tests were made using the hardware clock built-in to the processor. No attempt was made to calibrate the timing accuracy of this device, but the manufacturer's specifications claim accuracy to 1 μs.

TABLE 6.2 Arithmetic Benchmark Results for PPC750

Operation	Time (ns)
Double*Double	3
Double/Double	98
Float*Float	2
Float/Float	51
Int*Int	3
Short*Short	9
Double+Double	2
Float+Float	2
Int+Int	6
Short+Short	6
Double*Float	24
Double*Int	61
Float*Int	56
Double+Int	60
Double+Short	60

6.2.2 OSE Benchmarks

It is important to separate the timing overhead caused by GRRDE from the timing overhead added by the operating system itself. Table 6.3 summarizes OSE performance measurements published by ENEA [Liljedahl & Lillieskold, 1999]. These tests were performed on a slightly slower PowerPC750 processor than those used in GFLOPS.

6.3 Characterization of Time-Triggered Services

This section examines the run-time testing of the time-triggered subscription services. We wish to gain an understanding of the temporal behaviour of the system under different loading conditions. One of the primary applications of time-triggered subscriptions is that of digital control systems. Performance and stability of these controllers is highly dependent on the temporal accuracy of sensor and actuator handling [Marti, et al, 2001]. Thus, it

TABLE 6.3 OSE Performance Measurements

Operation	Time (μs)
Context Switch	1.6
System Call Overhead	.38
Semaphore: Wait	.52
Semaphore: Signal	.52
1-Byte Signal, Intra-Segment	2.2
1KB Signal, Intra-Segment	2.2
100 KB Signal, Intra-Segment	2.2
1-Byte Signal, Inter-Segment	3.1
1 KB Signal, Inter-Segment	5.3
100 KB Signal, Inter-Segment	570
1-Byte Signal, Network (TX)[a]	3
1-Byte Signal, Network (RX)	7
1 KB Signal, Network (TX)	9
1 KB Signal, Network (RX)	45

a. Network tests based on 10Base-T network.

is important to measure both the accuracy of the periodic message delivery and its variability.

6.3.1 Test Methodology

The test application for this experiment consisted of two GRRDE modules (Figure 6.1). The service module provides sample time-triggered subscription services. The test module created client processes for the server and logged timing information about the dispatch performance. Tests were conducted on a single processor, and across the network using multiple processors.

The service module is quite simple. It consists of only the GRRDE shell and a single dispatch function. The dispatch function is parameterized to allow the specification of an arbitrarily large dispatch signal. Upon triggering, the function allocates a signal buffer of

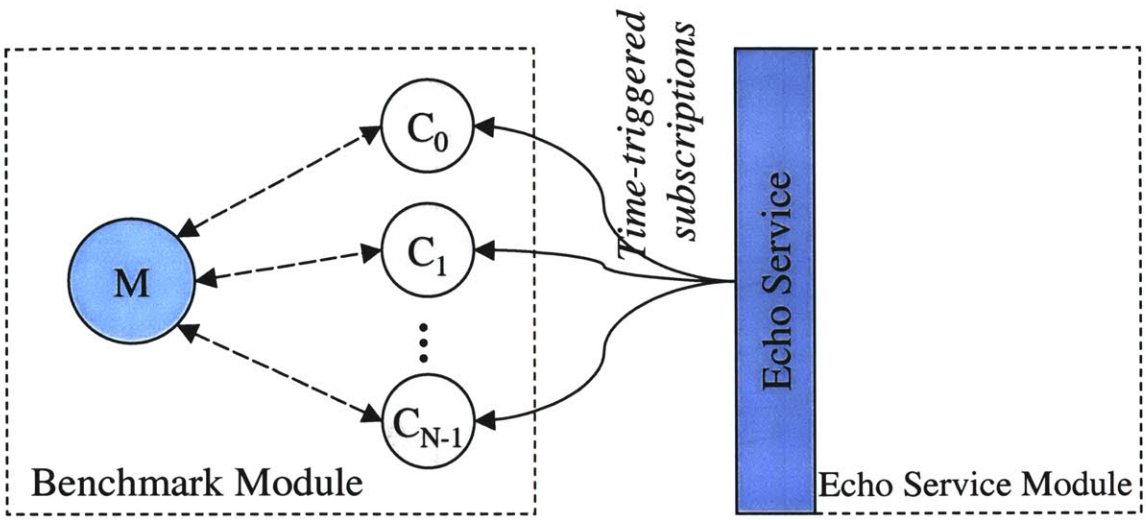


Figure 6.1 Test Setup for Time-Triggered Services.

the requested size and then send it to the destination process. No data was added to the buffer since the objective of the test was to measure the behaviour of the dispatching and delivery mechanisms. User-created processes were not needed in this module.

The test module was a little more sophisticated. A master process oversees the test progress. Each cycle, the master processes creates a number of client processes C_i. The master then informs the clients of the test parameters. Once initialized, the clients contact the echo server and request contract delivery. Tests were conducted to assess the effects of: subscription period (p), number of clients (N), synchronization of clients, message size (s), and differential prioritization of the clients.

The quantities measured are shown in Figure 6.2. Each test was of fixed length and included 1000 measurement intervals ($M = 1000$). Measurements were conducted independently by each client. The primary measurement was the dispatch arrival time, t. If the k-th message arrives at time t_k then the k-th inter-arrival time, τ_k, is:

$$\tau_k = t_{k+1} - t_k \tag{6.3}$$

and the mean period is:

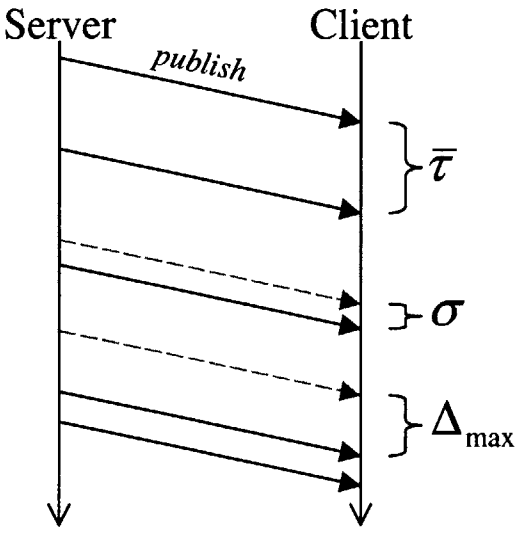


Figure 6.2 Timing diagram for time-triggered testing.

$$\bar{\tau} = \frac{1}{M}\sum_{k=0}^{M-1}\tau_k \tag{6.4}$$

The variance of the inter-arrival time was estimated from

$$\sigma^2 = \left(\frac{1}{M}\sum_{k=0}^{M-1}\tau^2\right) - (\bar{\tau})^2 \tag{6.5}$$

Finally, we also keep track of the maximum deviation Δ_{max} from the mean period and the interval during which this deviation occurred.

6.3.2 Test Results

The test space for these results is summarized in Table 6.4. The primary dependent variables were the number of clients, and the contract period. Timing tests were repeated to measure the timing between modules on the same processor, and between modules on different processors. Bandwidth limitations limited the test space for the large signal sizes. In the single processor tests, this limitation is due to memory bandwidth effects. Since the

two test modules reside in different memory segments, signal transmission requires a memory copy, thus limiting the maximum attainable rate. Network bandwidth has a much more pronounced effect. The theoretical bandwidth of 100Base-T ethernet is about 12.5MB/s. At that rate, the transfer of a 64KB signal buffer will take a minimum of 5.25ms.

TABLE 6.4 Summary of Test Space

Parameter	Range
Period	1 ms - 105 ms
# of clients	1 - 35
Signal Size	32 Byte, 64KB
Client Priorities	homogeneous, heterogeneous[a]

a. one high priority client

Single Processor Test Results

Across the entire spectrum of tests, the average inter-arrival time matches the requested period very precisely. Aggregating the percentage deviation, δ, over all tests and all reporting clients gives the data shown in Figure 6.3. This quantity is calculated from the reported value of $\bar{\tau}$, and the requested period:

$$\delta = \frac{p - \bar{\tau}}{p} \tag{6.6}$$

The data are tightly clustered. That the results reflect a wide variety of test conditions, suggests that the OSE and GRRDE timing characteristics are very good. Consideration of the shape of this histogram suggests a small tendency to undershoot the requested time interval. The cause of this bias is unclear, but may be attributed to the OSE timing routines. In any case, the deviations are small, even for the fastest periods, where the deviation of the average is less than 1µs. Since the mean accuracy of the subscription intervals seems very good, it is unlikely to create any problems when predicting system behaviour.

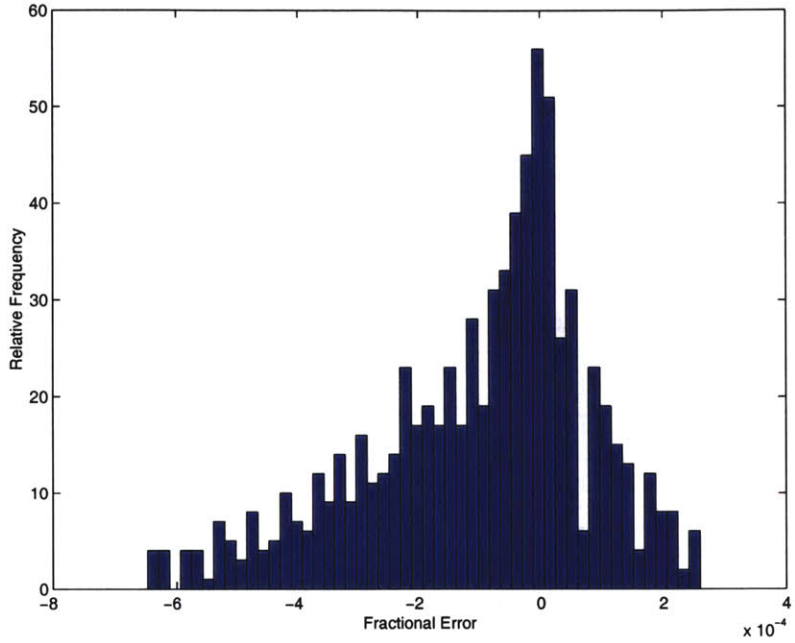


Figure 6.3 Fractional period deviation (Aggregate results from all tests).

Net results for the single processor tests are shown in Figure 6.4. This graph shows the results from the synchronized tests. During the synchronized tests, all the client processes attempted to initiate their contracts at the same time. During the un-synchronized tests the contract requests were distributed in time throughout a contract period. We expected the un-synchronized results to show less timing variation than the synchronized tests due to the effects of contention for the processor. Qualitatively, the effects observed were smaller than expected. Since the differences are difficult to distinguish on three-dimensional plots, the two types of testing are only shown together in selected two-dimensional slices.

Although processor synchronization did not have a great effect on the observed timing jitter, the prioritization of the processes was important. This effect is clearly apparent in Figure 6.5. This figure shows detailed results for the fastest (1 ms) subscriptions. Variability in arrival time is governed by three factors. First, the user must contend with inconsistencies in the OSE timing services. We cannot do anything to improve these results, but evidence suggests that the initial publish stimuli ($tick_i$ message) are generated very accurately. The second source of variability is within the GRRDE components. This is the

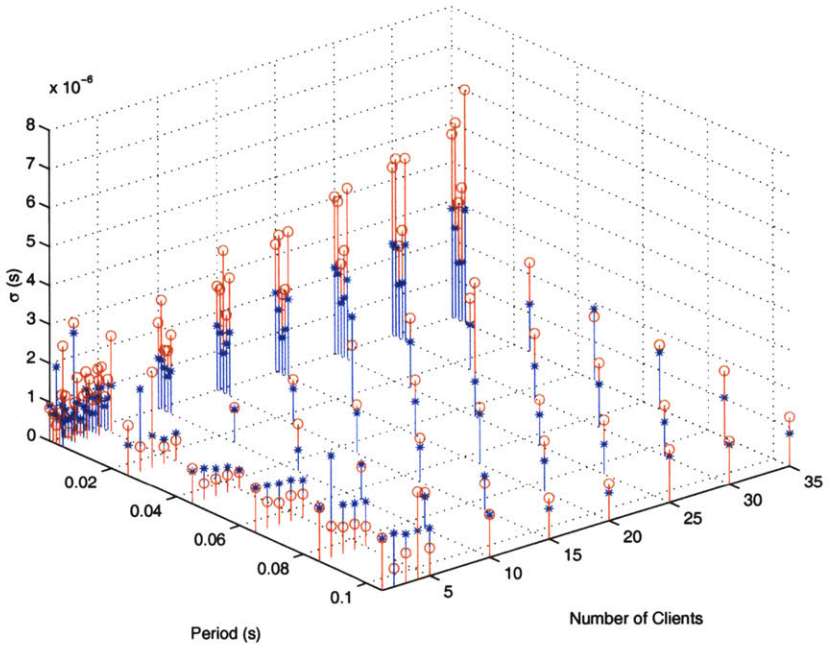


Figure 6.4 Standard Deviation (σ) of message delivery time. Low priority processes are marked with circles (red). High priority results are marked with stars (blue).

quantity that we wish to measure. Ideally, it will be as low as possible. The last source of variability is due to processor contention. We would like to be able to adjust system responsiveness so that critical processes experience less jitter than less important ones. In an RTOS, the fastest processes are usually the most time critical, and will usually have the highest priorities[1]. At present, GRRDE does not recognize any subscription prioritization. Therefore, if two subscriptions receive their stimuli at the same time, GRRDE will dispatch them in arbitrary order. Once the signals are generated, however, high priority processes will take precedence in their reception. Thus, high priority processes will preempt low priority ones, if they receive their publish messages at the same time. Figure 6.5 demonstrates that CPU contention accounts for over one third of the system timing variability for small signal sizes. This amount of timing jitter can be controlled through process prioritization. The remaining jitter is inherent to GRRDE and OSE. Examining the jitter

1. For a rate-monotonic system. See [Liu & Layland, 1973] for details.

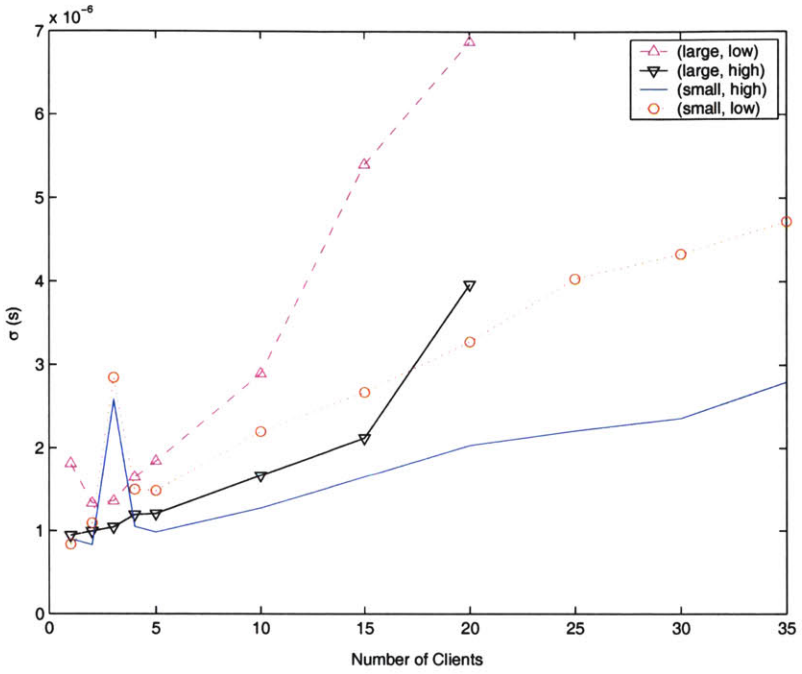


Figure 6.5 Standard Deviation of Publish Delivery (1ms period). Graph shows both high/low priority results, as well as those for large/small signal sizes.

observed with only a single client suggests that the operating system contributes a jitter of about 1µs. Therefore, the *internal* contribution from thirty-five mutual interacting contracts between GRRDE contracts is about 1.8µs (i.e. difference between the curves). Considering that the fastest timing loop that OSE can support with its built-in timing services is 1000Hz., such a small amount of jitter is unlikely to affect performance.

Memory bandwidth limitations can be seen in the curves for the large signal contracts. When the number of clients in the system is low, the observed jitter is higher than that for the smaller signals, but the slope is roughly the same. With fifteen clients, the high priority process is still fairly normal but the low priority performance is showing effects of backlog. With twenty clients the increased jitter is apparent in both curves. Beyond this point, the necessary throughput exceeds the computer's capacity.

The small signal tests experienced an anomalous spike in jitter for the three client test. This result is repeatable for a given ordered set of tests. Oddly, changing the testing order

will cause the reported values to change. The results for the three client test will lie more in line with the other values, but a similar anomaly will occur in another location. Although a definitive explanation for this behaviour was not found, we suspect that it is a consequence of the stack buffer recycling and allocation mechanisms in the OSE kernel. These routines are triggered by process creation or termination, such as might occur between tests. We suspect that a similar phenomenon is to blame for the drop in observed contract jitter between subscription periods of 3-5 ms.

Global test results for large signal sizes are shown in Figure 6.6. All tests save for some of the 1 ms subscriptions were completed satisfactorily. It is evident that the high traffic, short-period testing introduces a moderate amount of jitter, but once the periods become slower, the jitter is fairly insensitive to both the number of clients and the contract period. At high speeds, the observed jitter is higher than the small signal case, but not exceedingly so. This suggests a fairly robust and deterministic memory copying capability.

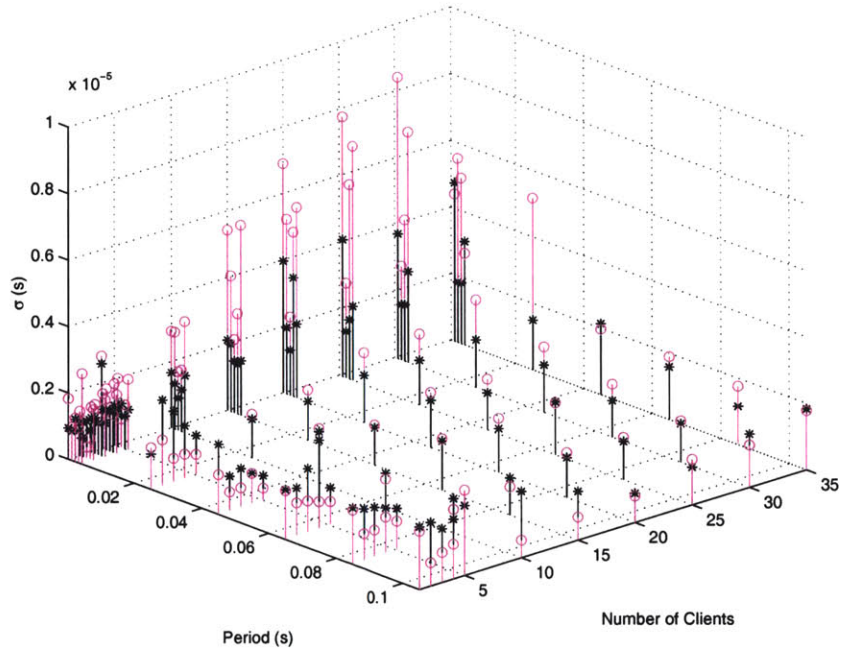


Figure 6.6 Standard deviation for large signal contracts. Low priority processes are marked with circles (maroon). High priority results are marked with stars (black).

As mentioned earlier, the effect of synchronization was less pronounced than expected. Characteristic results are shown in Figure 6.7. For the high priority contracts, the effects are minimal. The low priority clients exhibit some synchronization dependence over the range of subscription periods, but the effects are relatively minor.

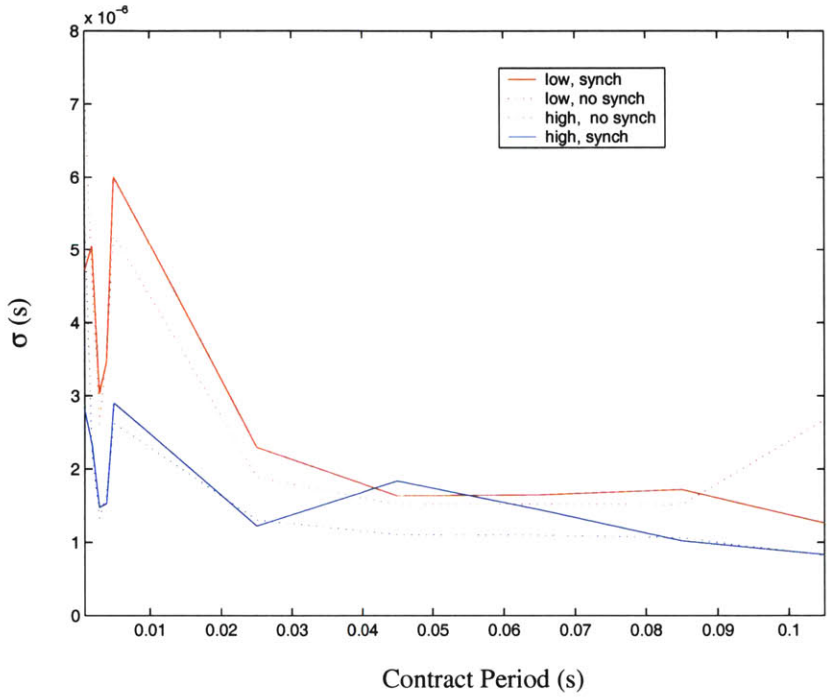


Figure 6.7 Effect of Synchronization (35 Clients).

Figure 6.8 shows the effect of subscription period on contract jitter. Beyond the initial range of relatively fast contracts, the jitter contribution is fairly insensitive to the period of the contracts. This agrees with the formal modelling from the previous chapter, which presented a dependence on the number of clients, but not on the period[1]. The test results seem somewhat erratic from one point to the next, but the gross shape of the curve agrees with our modelling and intuition.

1. Since IOA actions execute atomically, process interactions are difficult to model explicitly.

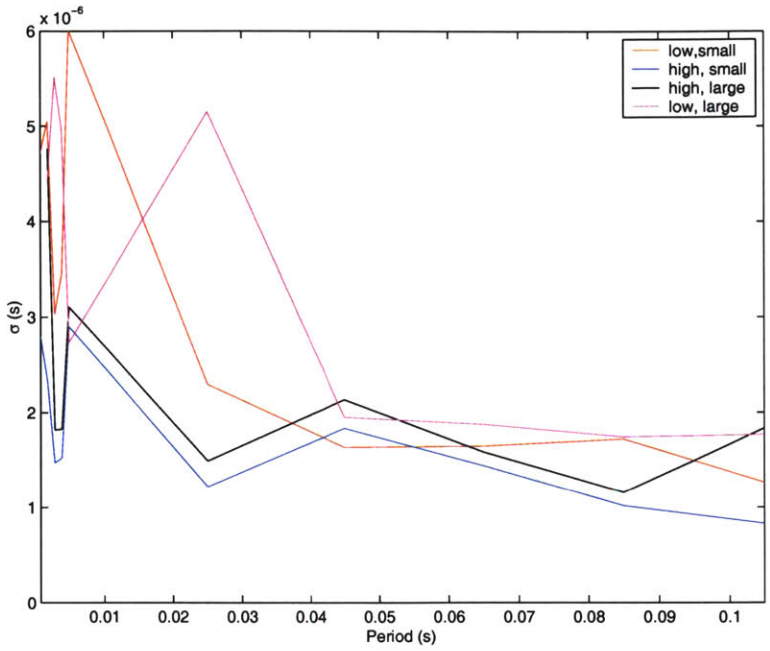


Figure 6.8 Period dependence (35 Clients).

Most of the discussion of jitter up until this point has addressed the jitter variance (actually the square-root of the variance, or standard deviation). If we assume that the jitter distribution is normally distributed, roughly 67% of the samples lie within $\pm\sigma$ of the mean. Also important in the study of GRRDE performance is the maximum observed jitter. Typical results are shown in Figure 6.9. The plots showing synchronized results and large signal sizes are very similar and have been omitted for the sake of brevity. Most areas of the figure show that the maximum jitter lies in the range of 10-25 μs. Elsewhere, particularly for short period contracts, much higher values are sometimes observed, reaching almost 100 μs.

From an embedded software engineering perspective, these seemingly anomalous jumps in timing are disturbing. Real-time software, above all else, must be predictable. These timing glitches may be enough to cause concern over control system stability. We can be somewhat reassured, however, if we consider the locations of the maximum deviations. Figure 6.10 shows a histogram of aggregate results over all the tests that were run. If these

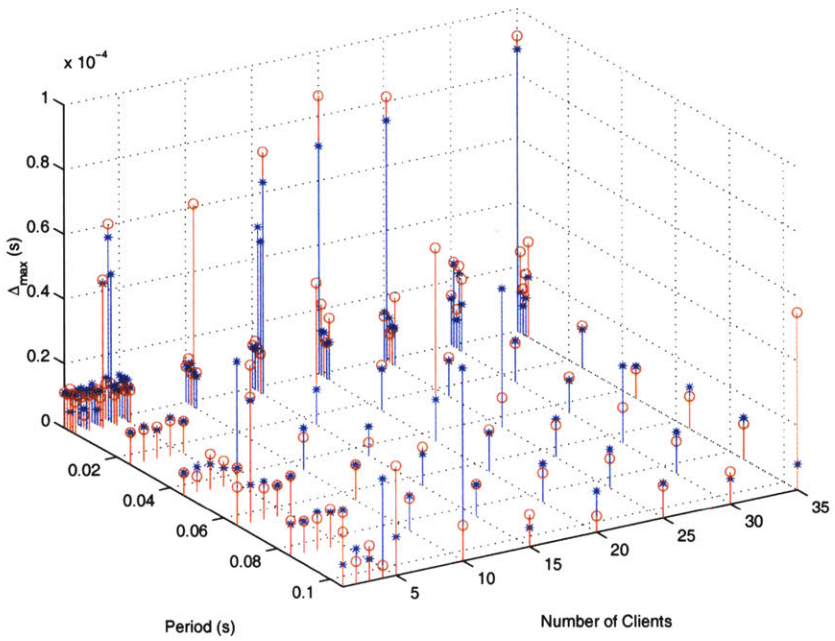


Figure 6.9 Maximum Deviations (Small signals, no synchronization). Low priority processes are marked with circles (red). High priority results are marked with stars (blue).

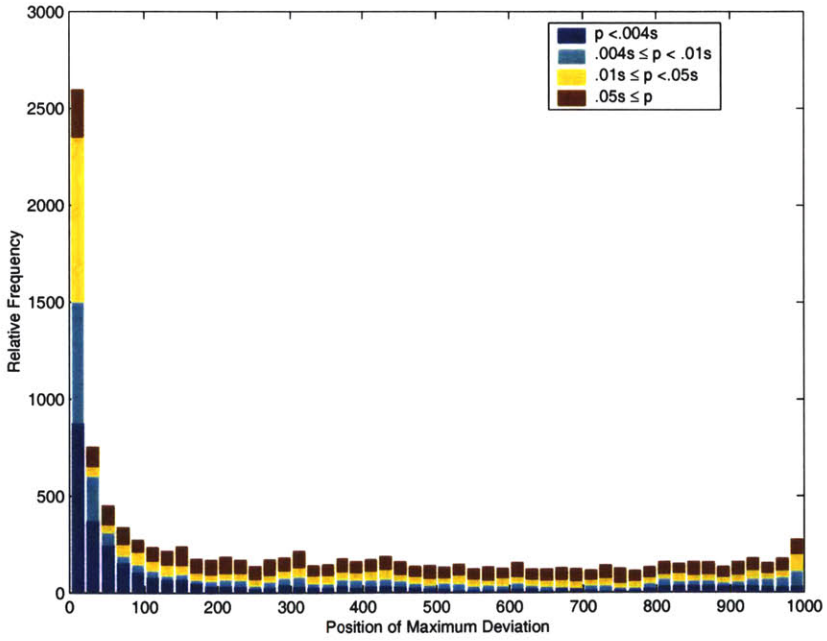


Figure 6.10 Distribution of maximum jitter observations. The pronounced bias suggests the large role of contract start-up transients.

spikes in timing uncertainty were truly random, we would expect to see a fairly flat distribution. The observed bias in the data indicates that startup effects are responsible for many of the reported maximum values. Unfortunately, many of the large glitches are not found at the start of a test. We suspect that they may be caused by preemption by OSE kernel processes.

Networked Tests

Once the single processor timed contract test was completed, the same test modules were loaded onto separate machines and the clients were forced to remotely access the service module. In this test we examine how the GRRDE time-triggered subscription services

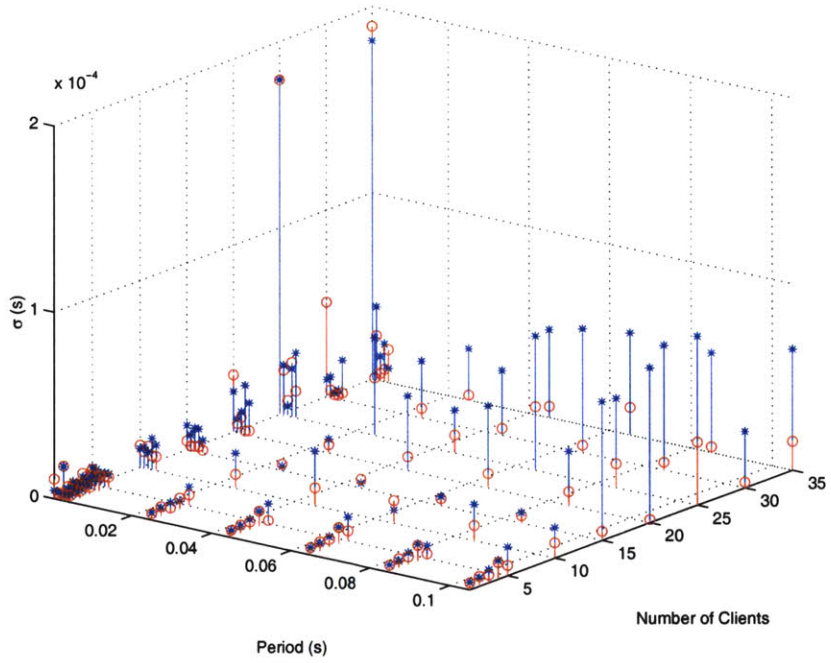


Figure 6.11 Network performance for small, synchronized subscriptions. Low priority processes are marked with circles (red). High priority results are marked with stars (blue).

behave in a networked environment. Of principal interest is any additional jitter, rather than absolute delay and throughput.

Networked signal transfer using OSE takes three steps. First, the transmitted signal must be copied from the source memory pool to the kernel pool. Second, the source Link-Handler process transmits the signal data to the destination link-handler over the network. This step may involve the 'packetization' of the signal, and many internal operations. The signal is reconstructed in the kernel pool of the destination CPU. The third step is to copy the signal from the kernel pool to its destination module. When network traffic is high, the temporal and memory performance of the system is governed by the relative process priorities of the destination link-handler and destination processes.

If the destination process priority is higher than the Link-Handler's, the destination processes can process the signals as soon as they arrive over the network. Unfortunately, this configuration preempts the link-handler and degrades the total throughput and network jitter. Alternately, if the prioritization of the processes is reversed, the link-handler can operate efficiently, but messages will become backlogged in the kernel and destination pools, since the client processes cannot process them while the link handler is active. This substantially increases the demand for memory since duplicate storage must be allocated.

Test results for synchronized and un-synchronized clients are shown in Figure 6.11 and Figure 6.12. Nominal values are typically less than 20μs for fast contracts and 10μs for slower values. Very fast contracts show the worst behaviour with some jitter results near 200μs. It is unclear why high priority contracts experience *worse* jitter than low priority subscriptions. This trend is particularly evident in synchronized tests. We suspect that it is due in part to context switching and CPU contention between the link-handler and client processes. Tuning the OSE network performance is a possible solution to this problem, but extensive evaluation of this possibility has not been evaluated.

Maximum jitter observed in these timing tests remained quite modest. Figure 6.13 shows the dependence on number of clients for 1 ms contracts using small signals. Three of the

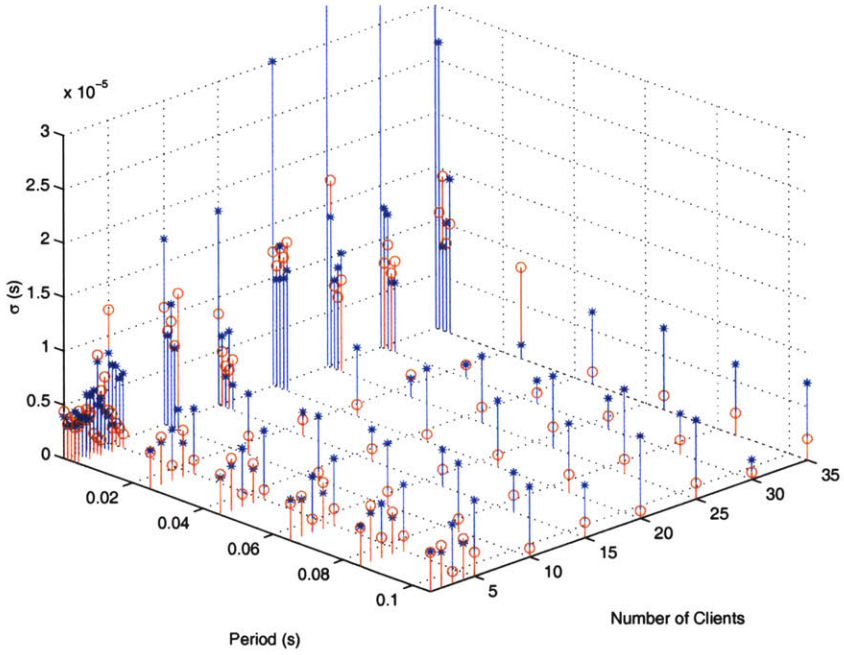


Figure 6.12 Network performance for small signal, un-synchronized subscriptions. The truncated values are about 2e-4 s. Low priority processes are marked with circles (red). High priority results are marked with stars (blue).

curves show values growing to about 500ms, or almost half of a subscription period. The large jump observed by the low-priority un-synchronized clients is almost certainly related to network backlog.

The large signal tests were severely restricted by the bandwidth of the network. Flooding the network with large packets overwhelms the link-handler. Systems transferring large quantities of data over the network should either implement end-to-end flow control or include enough memory in the system pool to handle the potentially large backlog of packets.

Figure 6.14 shows the dependence of jitter on contract period. Even with only four clients, many of the faster tests exceeded network bandwidth. The curves in the above graph show no clear trend, but seem comfortably bounded by a standard deviation of 10μs.

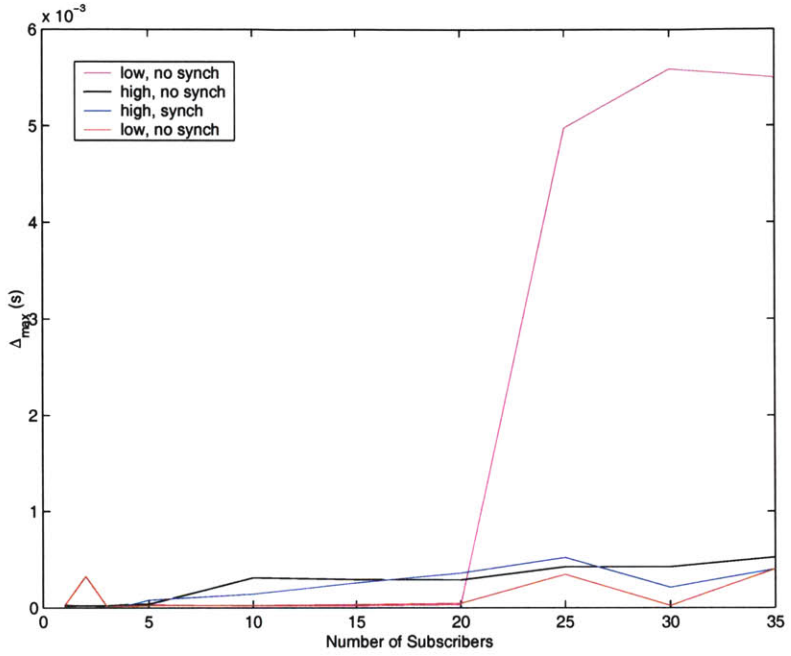


Figure 6.13 Maximum network jitter, small signal test, 1 μs subscriptions.

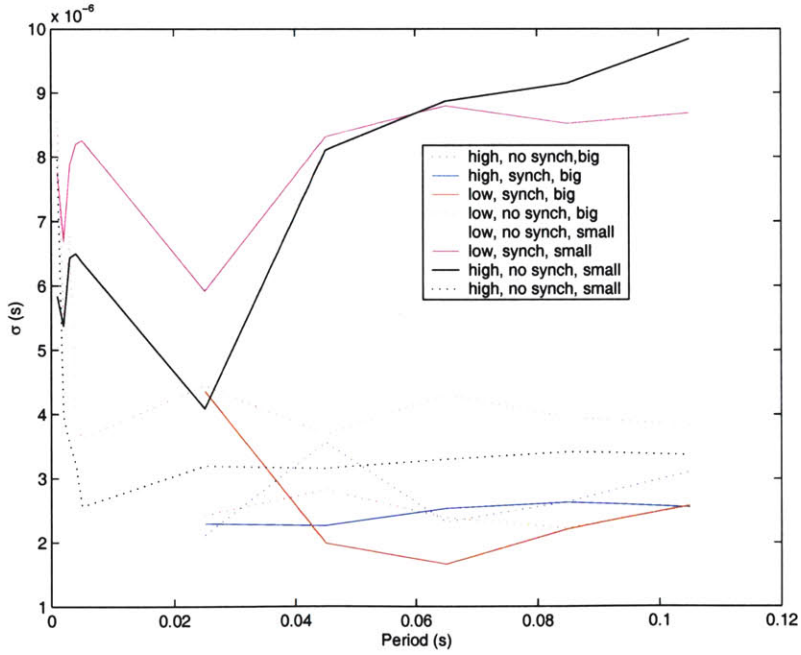


Figure 6.14 Effect of period on network jitter (4 clients).

A fundamental characteristic of middleware is its network performance. Since GRRDE is a form of middleware, characterizing this performance is important. For most of the tests described above, the observed jitter was below 20μs. The expected performance degradation from this level of jitter is minimal. Since the GRRDE dispatch mechanism operates on a single processor, regardless of the client location, we expect that most of the observed performance degradations can be attributed to the OSE networking components rather than GRRDE.

Several factors make network performance estimates difficult. First, there is the question of our network protocols and architecture. Both the physical network (100Base-T) and the network protocol used by the OSE link-handler (user datagram protocol, or UDP) are not designed to provide real-time performance guarantees. Although both are engineered to provide good average-case performance, they make no guarantees about worst-case behaviour. Use of a synchronous network such as MilStd-1553 would make performance more predictable. The second difficulty lies with tuning the performance of the link-handler itself. If the present level of jitter exceeds the requirement of the user's system, adjustments to the OSE link handler components may solve the problem.

6.3.3 Parameter Estimation

In the previous chapter we derived equations describing bounds on system jitter. We would like to correlate those results with the measurements we have made of actual performance testing. Let us take Eqn. 5.31, for example:

$$\begin{aligned} pub\_jitter \;=\; & timer.jitter + r\_disp\_bnd.ub + r\_exec\_bnd.ub + d + r\_return\_bnd.ub \\ & + (N-1)\cdot max(r\_exec\_bnd.ub,\, w\_exec\_bnd.ub) \end{aligned} \tag{6.7}$$

Each of these parameters represents the upper bound on a particular step in the dispatch algorithm. Unfortunately, the equation is a little too precise to be useful. The problem arises when we consider that the General Timed Automata model cannot explicitly model processor contention. Instead, we must separately determine the maximum blocking and execution time and use the resulting value as an execution time-bound. Thus, determining

the total blocking time requires knowledge about how the whole simulation is composed. Also, most of the important parameters in the above formulation (e.g. *r_exec_bnd, w_exec_bnd*) are specific to the data service.

A more intuitive approach, is to simplify the above equation. The important dependency we wish to preserve is the scaling with the number of subscribers. This yields the simple linear relation:

$$pub\_jitter = x_1 \cdot N + x_o \tag{6.8}$$

Now we are left with the matter of fitting data to this curve. Instead of using the Δ_{max} data that we gathered, we fit the curve to the standard deviation measurements. We justify this decision based on two reasons. First, the quantity that we are trying to determine is the *upper bound* on jitter. In any given test there is no guarantee that we will observe this maximum value. Second, the Δ_{max} data has a number of spurious values that we suspect were caused, not by GRRDE, but by other system activities. Consider Figure 6.15. The figure shows the subscriber dependence for standard deviation, σ. Individually, the data are rather noisy, but averaging the curves gives the heavy black line. This curve is fairly linear. A straight line fit to this curve gives $x_0 = 1 \times 10^{-6}$ and $x_1 = 1 \times 10^{-7}$. If we assume that the jitter is normally distributed, a suitable multiplier can be applied to the coefficients to give an adequate margin of safety (e.g. four-sigma, five-sigma, etc.).

Similar curves can be derived for large signals and un-synchronized subscriptions. However, the usefulness of actual numbers is rather limited. For the single processor test, we contend that the primary contributors to delivery jitter are the dispatch function jitter (*r_exec_bnd*), and the mutual process interactions which cause the linear scaling. Measuring the dispatch function jitter in isolation gives a standard deviation of 1.05μs. This value is consistent with the observed jitter in the full test when $N = 1$. Thus, publication jitter is dominated by the user's code and not the internal mechanisms of GRRDE. Consequently, the above derivation should not be interpreted as a prediction of absolute jitter, but rather a method that developers may use to analyze their own simulations.

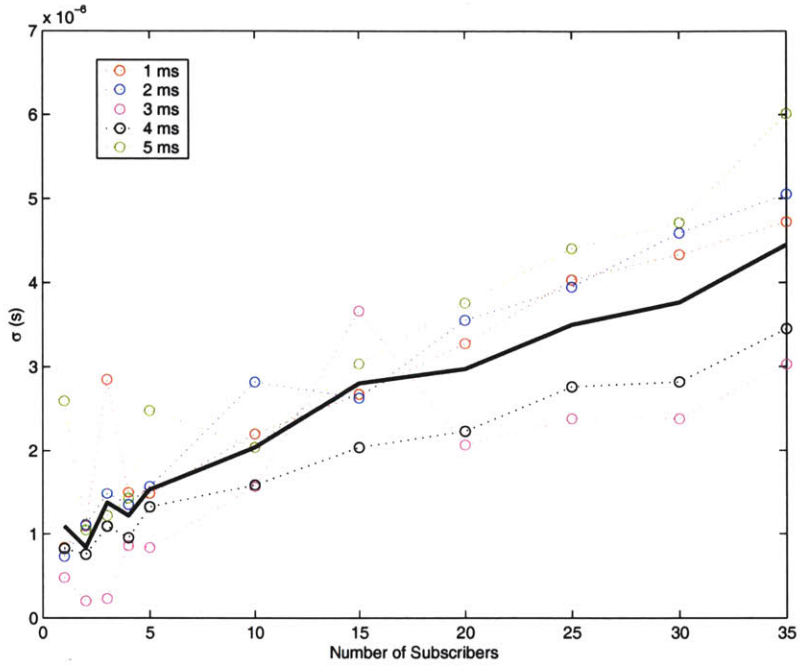


Figure 6.15 Small signal standard deviation (low-priority, synchronized). The heavy line is the average of the five component curves.

The network tests are very noisy and do not lend themselves well to similar analysis. As mentioned earlier, we suspect that the variability has more to do with an ill-tuned link-handler and a non-deterministic network, than any limitations of GRRDE itself.

6.4 Change Triggered Subscription Testing

Measuring timing behaviour is an important part of characterizing GRRDE's performance. These services are less likely to be used directly in control loops, so latency is not as much of a concern. However, like any embedded component, efficiency and determinism are both virtues.

6.4.1 Test Description

To complement the temporal characterization of the GRRDE time-triggered subscription services, similar tests were performed on the change-triggered mechanism. The test formulation (Figure 6.16) was very similar to the previous example. Two test modules are

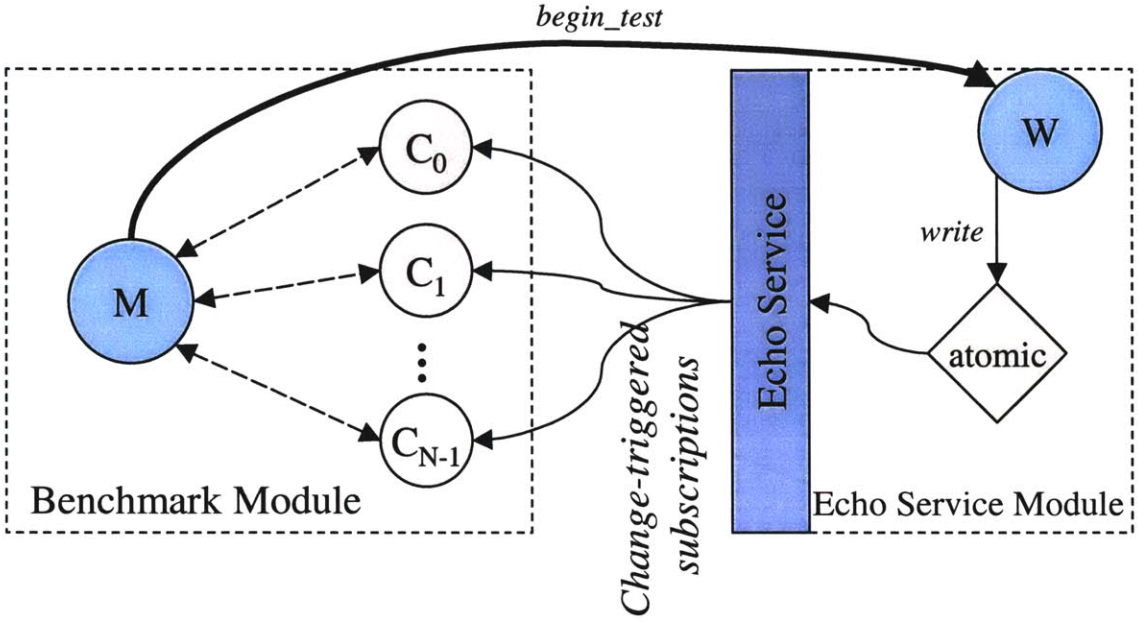


Figure 6.16 Test formulation for change triggered services.

necessary for this test. The master test process, M, creates and initializes a number of client test processes, C_i. Each of these client processes subscribes to the change triggered service provided by the echo module. This module is more sophisticated than the module used for time-triggered testing. Within the module is a writing process, W, and a $atomic2$-type atomic object. Once the client processes have started, the master test process signals W to begin the test. This stimulus causes W to begin periodically writing to the atomic variable. For each cycle, the writing process will get the current time from OSE, and write this value to the atomic object. The signal dispatched from the echo module contains this time record, and an additional 'dummy' payload of arbitrary length. When the clients receive the message they record the current time and compare it to the time of writing. The timing diagram for these measurements is shown in Figure 6.17.

Several important features distinguish the change triggered testing from the time-triggered tests. First, tests were only conducted on a single computer. Since the interface to network transport is external to the GRRDE dispatch mechanism, the added jitter should be similar to that encountered in the previous tests[1]. Second, the quantity that we are measuring is

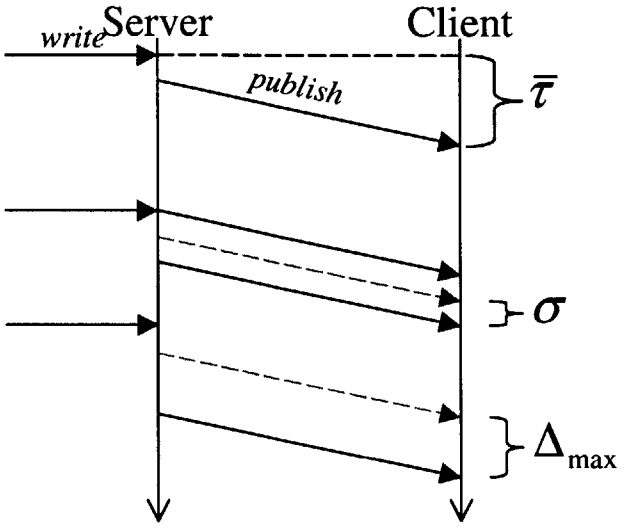


Figure 6.17 Timing measurements for change-triggered contracts.

qualitatively different. This test measures the time elapsed from the write invocation, to the delivery of the message to the client. In each trial, we calculate the propagation interval τ_k. The variance of this quantity is calculated identically to Eqn. 6.5. Once again we record the position and value of the maximum deviation from the mean.

6.4.2 Test Results

This test suite is more compact than the one employed in the preceding section. Contract period variations would be meaningless in a change triggered system, as would the effect of synchronization. Both of these considerations are ignored. The tests vary the number of clients in the system as well as the process prioritization and signal size.

Each time W writes a new time value to the atomic object, a dispatch is initiated. A publish message is then generated for each active subscription. Although this generation process has been modelled as a simultaneous operation, the operations are performed

1. Additionally, distributed time measurement is difficult at the accuracies that we wish to measure without using additional instrumentation and clock synchronization protocols. Using a single computer to make the timing measurements ensures a consistent view of elapsed time.

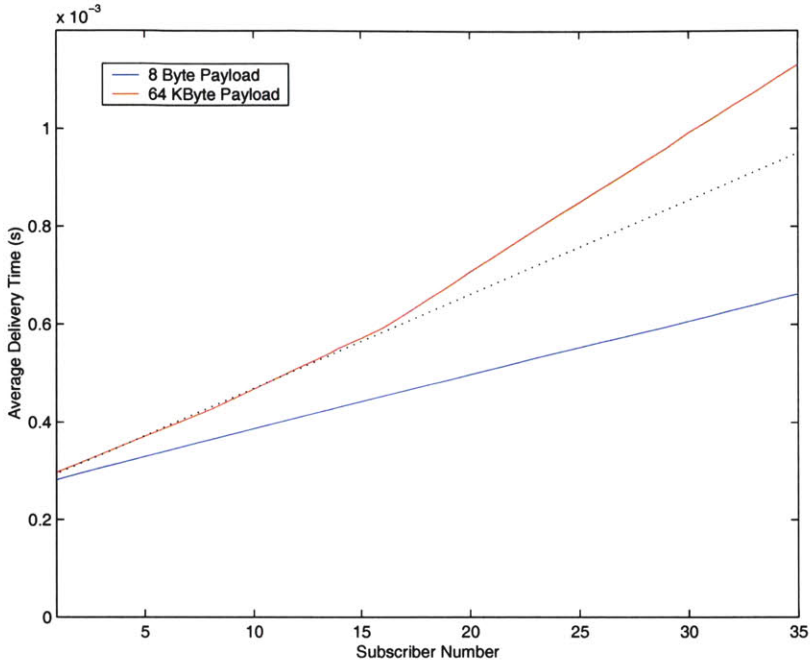


Figure 6.18 Dispatch delay as a function of subscription index.

sequentially. This yields the delivery time curves shown in Figure 6.18.This graph shows the average arrival time for each client process. The GRRDE dispatcher generates the publish message in the order in which the subscriptions were requested. Thus, for a fixed set of subscribers, the order in which the dispatch messages are generated is deterministic. The delivery time shows a fairly linear increase as a function of subscriber number. The knee in the large signal (65kB) curve suggests memory bandwidth effects coming into play. A linear data fit to the small signal (8B) curve gives the delay relation for process i (in μs):

$$\tau = 275 + 11i \tag{6.9}$$

and for the initial portion of the large signal subscriptions:

$$\tau = 275 + 20i \tag{6.10}$$

This delay is quite substantial, even for relatively low numbers of subscribers. The extent to which this would affect system performance requires additional investigation. The rea-

sons for this delay are unclear, but warrant further study. It should be possible to determine whether such delays are unavoidable consequences of the dispatch mechanism or whether they are caused by inefficiencies in implementation.

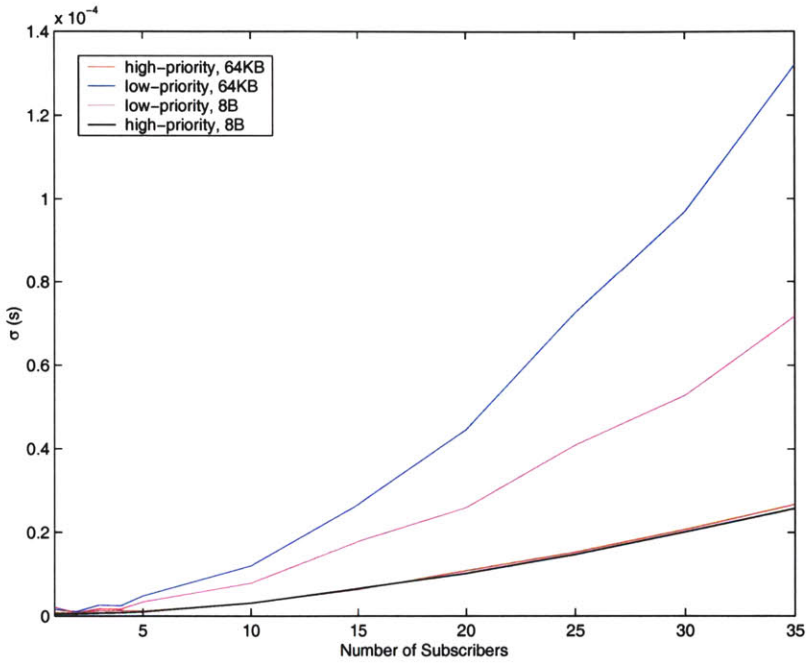


Figure 6.19 Standard deviation of change-triggered jitter.

The one-sigma and maximum jitter results are shown in Figure 6.19 and Figure 6.20. The standard deviation of the delivery time appears to scale non-linearly with the number of subscribers. This might be ascribed to unmodelled CPU contention between client processes. High-priority process jitter seems quite reasonable; with thirty-five subscribers the jitter is less than 30μs. The low-priority clients fare somewhat worse, with values of 70μs or 130μs, depending on the signal size. Maximum jitter appears to scale linearly with the number of subscribers. This agrees with our model derived from the formal analysis (Chapter 5). The maximum observed jitter for both large and small, high-priority signals is approximately 250μs. Low-priority results are substantially higher. The small signals still appear to be linear, but the large signals are beginning to show non-linear effects.

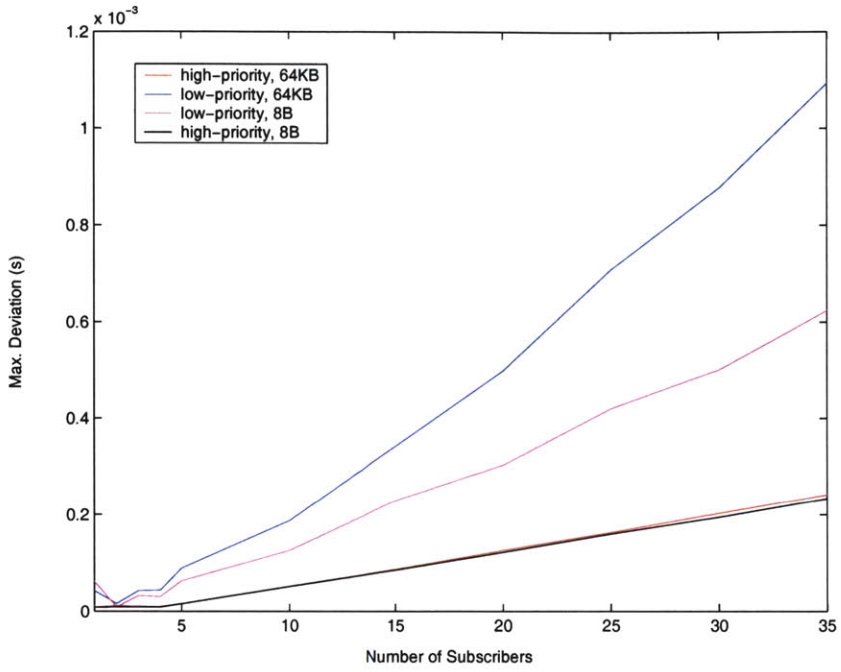


Figure 6.20 Maximum jitter, change triggered subscriptions.

Although not reflected in the figures, examination of the raw data suggests that most of the time the maximum deviation from the mean occurs in the negative direction; i.e. the dispatch takes less time than the mean. In fact, the observed maximum positive deviation is rarely much more than the standard deviation. Examining the location of the maximum deviation (Figure 6.21), suggests that transient behaviour is to blame for these effects. At the start of a test, GRRDE is still processing contract requests. At the end of the tests, some processes may finish before their peers. Presented with a sudden drop in the number of active contracts, GRRDE can process the remaining dispatches much more quickly.

6.4.3 Parameter Estimation

The test regime for change triggered contracts was somewhat more benign than encountered in the time-triggered case. Timing results for very fast contracts showed signs of backlog. Although it is important for the designer to consider the net system load in different operating conditions, it is good design practice to keep cyclic utilization to levels that the processor can handle without being overloaded. Since the invocation rate of the

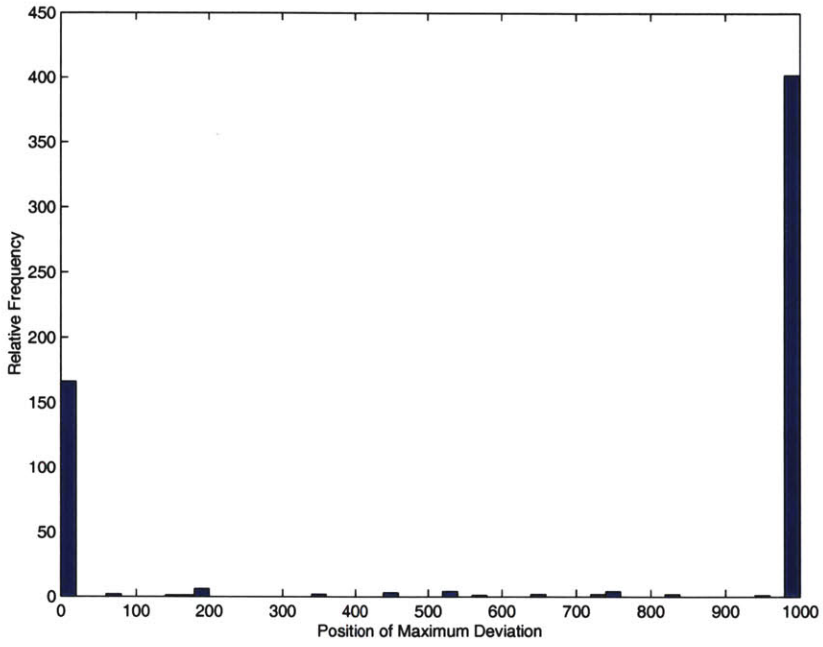


Figure 6.21 Location of maximum jitter. Transient effects at contract start or completion account for most of the maximum values.

change triggered contracts was fairly slow (100Hz), measurements taken were of nominal, rather than overload, jitter.

The publish jitter Δ_{pub} is given by Eqn. 5.41:

$$\begin{aligned}\Delta_{pub} &= r\_disp\_bnd.ub + (N_s - 1) \cdot r\_exec\_bnd.ub \\ &\quad + r\_exec\_bnd.ub + notify\_time.ub + dispatch\_bound.ub\end{aligned} \tag{6.11}$$

and the write commit time is given by Eqn. 5.39:

$$WriteCommitTime.ub = w\_disp\_bnd.ub + (N_w - 1) \cdot (W + N_s \cdot R) + w\_exec\_bnd.ub \tag{6.12}$$

The sum of these two equations describes the expected scaling of the upper bound on dispatch jitter. Combining the scalar parameters, gives a simple linear equation in N_s:

$$\Delta_{total} = y_0 + y_1 \cdot N_s \tag{6.13}$$

Since these tests did not overload the processor, the observed maximum values can be fit fairly well to such an expression.

6.5 Comparison to Published CORBA Tests

A paper by Schmidt, *et al* [Schmidt, et al, 1997] discusses temporal performance comparisons of several middleware systems. The authors examine several implementations of the Common Object Request Broker Architecture (see Section 2.2.2) and discuss how the implementation architecture affects their temporal performance. Differences in testing architectures preclude direct comparison to our GRRDE results but the general trends and findings provide useful perspective.

Schmidt examined four CORBA implementations: IONA's MT-Orbix, Exersoft's CORBAplus, Sun's miniCOOL, and their own implementation, TAO. It is worth noting that the miniCOOL implementation is specially adapted for embedded applications and TAO was specifically designed for real-time performance. The study compared timing measured by client applications, in contact with a remote (networked) server. The server application performed a very simple calculation and returned the result. Test results measured two-way latency and jitter for high and low priority clients.

The testing revealed the best performing ORBs to be the miniCool and TAO implementations. Between these two finalists, TAO, fared substantially better than miniCOOL. The authors contend that the performance increase was due to a design specifically addressing real-time considerations. In contrast, miniCOOL was reengineered for embedded applications, rather than being designed so from the start. The reported latency for the TAO implementation was about 1.2-2 ms for both high and low priority clients. Jitter was reported as less than 1 ms for high priority clients and 5-11 ms for low priority clients.

Significant discrepancies exist between the testing environment reported in the study and GRRDE. Network technology, test computers, and operating systems were sufficiently different that a direct comparison of results cannot be made. Even so, we feel that on a

qualitative basis, GRRDE compares favourably. The time-triggered network tests, even if doubled to account for two-way effects, lie in the same range as the TAO results. Change-triggered subscriptions also exhibit latency and jitter comparable to TAO.

TAO has been cited by a number of sources [Bates, 1998] [Emmerich, 2000] as a promising example of middleware for real-time applications. At first glance, the temporal performance of GRRDE is similarly stable. While direct comparison would require hands-on testing of both products, and a more elaborate test regimen, GRRDE seems to be a solid competitor.

6.6 Summary

Correct operation of an embedded system relies on the predictable nature of its components. Therefore, before embedded software can be trusted, developers must satisfy themselves that it will remain deterministic within the anticipated range of system states. Designers choosing to adopt GRRDE for simulation or flight software development can be reassured by the verification of GRRDE's performance demonstrated in the last two chapters. The linear bounds on system jitter derived in the previous chapter appear in the results of the run-time testing. Although unmodelled non-idealities such as memory bandwidth and CPU contention have some effect on the results, the most common sources of maximum jitter are transient events, such as the start or end of the tests. This effect is to be expected in any preemptive multi-processing environment where aperiodic events can easily disturb the rhythm of the periodic processes. The effect of these situations can be evaluated through further testing and conventional real-time analysis. Conservative upper bounds on GRRDE performance exhibit both wide applicability and sub-millisecond jitter.

Chapter 7

SOFTWARE DESIGN USING GRRDE

Earlier chapters have explored the issues involved in designing individual modules for GRRDE. This modular approach is suitable for developing both the *flight software* and the *simulator* components that together form a GFLOPS simulation (Figure 7.1). Although module-level design is a vital part of the GRRDE approach, the larger considerations of simulation architecture are no less fundamental. In this chapter, we examine a number of issues that designers must address when defining the structure of large simulations.

We begin the discussion with a look at some of the ways in which functional blocks can be joined together as architectural elements. A simple control system example illustrates some of these concepts. Inter-connections between GRRDE modules require precise documentation of system interfaces. We have developed a structured approach to writing specifications for these interfaces that complements the GRRDE publish-subscribe services. In Section 7.2, we discuss several embedded considerations and ways in which traditional real-time analysis can be applied to GFLOPS simulations. Finally we propose ways to migrate a GRRDE-based software simulation to a hardware testbed or deployed system.

7.1 Architectural Considerations

Formulation of high-level system architecture is an essential part of software engineering. It is important to realize that this process is recognized as a separate mental exercise than

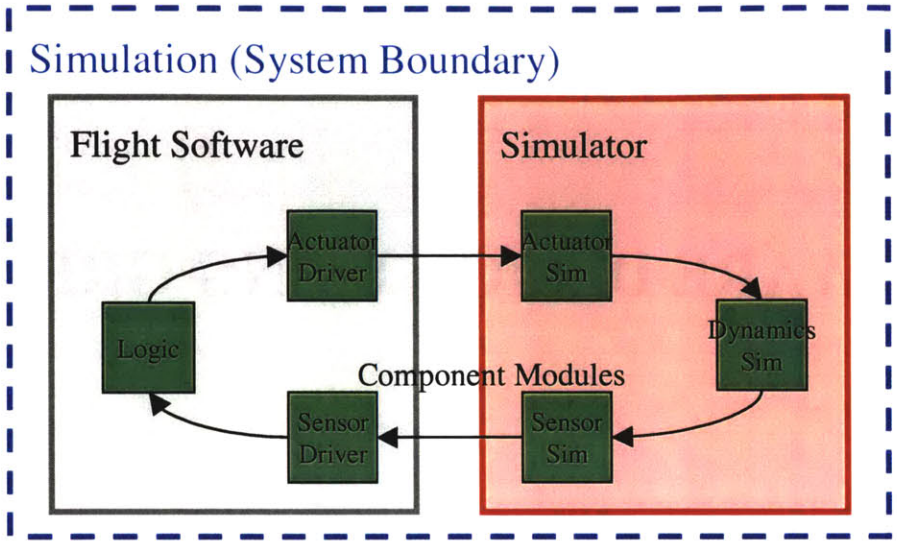


Figure 7.1 Typical simulation architecture. Flight software modules and simulator modules together form a simulation.

low-level algorithm development [DeRemer & Kron, 1976]. Having discussed in Chapter 4 the construction of individual GRRDE models, we now consider how these modules may be connected together to form complete simulations. This process supplements those stages of design involving functional decomposition and offers a variety of solutions to common embedded scenarios. It is not our intention to offer a categorical approach to simulation design. Instead, we offer a number of suggestions as to how some typical problems can be viewed from the GRRDE perspective.

We first discuss some ways in which modules can communicate. We may know from a superficial design that modules A and B share data x. We must still consider other aspects of the connection such as: "Who is responsible for the link?", or, "What drives the exchange of information?" These abstractions represent common elements from which the overall architecture can be assembled. GRRDE's information flexibility provides architectural capabilities that can be exploited as design and implementation matures.

7.1.1 Architectural Communication Elements

In Chapter 4, we discussed the state-mobility principles of the publish-subscribe services. The job remains of specifying how these services can be used to create effective simulations. Contracts are not the only acceptable way to pass messages between modules. It is frequently desirable to communicate in other ways; dispatch functions can be invoked without reference to a contract, and messages can be sent directly to or from individual processes. This section explores some of the variations commonly encountered with corresponding discussions of their appropriateness to different situations. A proposed notation scheme helps in communicating these ideas.

The classifications of inputs and outputs are based around several criteria. It is important to consider where the information comes from, which actors create the data pathways, and how messages are handled at the block interface. The principal communication types and their notations are shown in Figure 7.2.

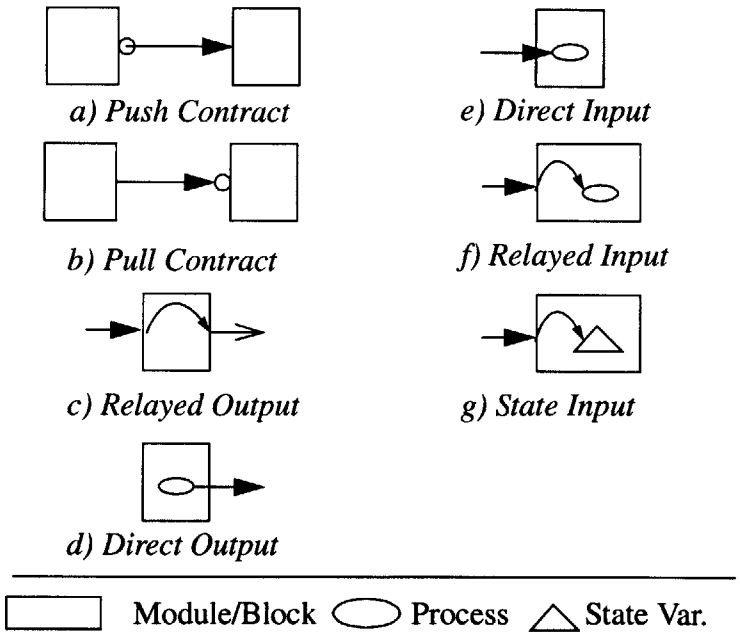


Figure 7.2 Input and output types.

Contract based output is the easiest to describe. With the GRRDE emphasis on state centric design, a large fraction of the IPC should be handled this way. The GSCA supports two primary types: push and pull. 'Push' contracts are initiated by a source that knows of a destination for its data. 'Pull' contracts in contrast are initiated at the destination. The information sink must know the name of the source service. Functionally, both types of contract operate identically. The only difference is the conceptual 'responsibility' for the information pathway[1]. The push-type contracts are often used in conjunction with a subsequent relay.

Some modules may act as simple filters or relays for incoming signals. An output module that performs pulse-width modulation is a possible example. A specific abstract command (e.g. a force) is thus converted into one or more reactive operations. In the above notation, the 'relay output' designation indicates an output that is in direct response to the receipt of an input. This special-purpose type of communication is usually found at the interface between simulator and flight software.

Module outputs can also come directly from a process. It is not uncommon to encounter situations where this manner of communication is the most intuitive. Delivering aperiodic commands to other modules is most easily envisioned through direct signalling. Also, some operating system services are accessed through a signal interface. Care must be taken however when designing simulations using direct messaging. Non-standard communication requires explicit interface specification, especially if handshaking or other protocols are involved.

Outputs from one module become inputs to another. As there are many forms of outputs so there are various forms of input. The simplest input is a signal that is addressed directly to a particular process within a module. To maintain encapsulation, it is strongly recommended that direct signal reception be applied circumspectly. Most incoming block traffic

1. This introduces the possibility of a third-party contract, initiated by neither the source nor the sink

should be relayed through the address process. The only time an active process should directly receive an outside signal is in direct response to a query that it has previously sent.

Inputs that are relayed through a block's address process (the phantom process mentioned earlier) can be treated in one of two ways. If the incoming signal represents the update of an input state variable, it is called a state input. Signals that are relayed directly to internal processes are denoted relayed inputs. Inputs of both types must be articulated in the block's input specification. When inputs are used to update a local copy of a continuous state variable, any old data is typically overwritten and there is no guarantee that two closely-spaced inputs will be processed individually by the internal logic. Relayed-inputs, in contrast, typically represent commands or queries that must be handled individually. Modules designed in this manner can take advantage of the built-in signal queueing functions of the operating system. The difference between these two techniques lies in the effect of inputs on block operation. State inputs do not directly affect the processor utilization of the destination module. The requirement that relayed inputs be processed individually means that the processing needs may be bursty.

Simulations are constructed by combining functional modules with communication elements. GRRDE provides developers with some flexibility in establishing these links. Contracts can even be specified at runtime. This flexibility is examined in the next section.

7.1.2 Information Flexibility

The combination of named service registration and the publish-subscribe communication services provides significant flexibility in simulation construction. Such flexibility aids the development process by promoting quick prototyping and allowing engineers to explore different information architectures. The extent to which a part of a simulation can be replaced by a different implementation of that segment defines the architecture flexibility. This freedom must be tempered by the requirements of embedded systems. Flexibility can be introduced to the extent that it does not create significant processing or memory overhead.

Interface definitions leverage the use of the registry service provided by the operating system. Both input services (e.g. an actuator interface) or output services (e.g. a sensor reading) can be registered. It is expected that registered outputs will be far more common than inputs. Since there is no arbitration mechanism to resolve duplicate names in the service registry, all registered interfaces should be unique. Similar labeling of related services (i.e. "Sat_1_Position" and "Sat_2_Position") is permitted (even encouraged), but must be managed by the user and does not carry special semantic meanings for GRRDE. These decisions enforce the need for alternate data sources or sinks to be handled explicitly by the functional modules.

Once started, a module should seek to establish the communications links for which it is responsible. This may consist of initiating contracts or simply looking up remote blocks. Most contracts will be started by source or destination blocks, but contracts may be initiated by a third party.

Information flexibility in GRRDE allows a data pathway to be replaced by another, if the external interfaces are maintained. This process is depicted in Figure 7.3. Starting from an

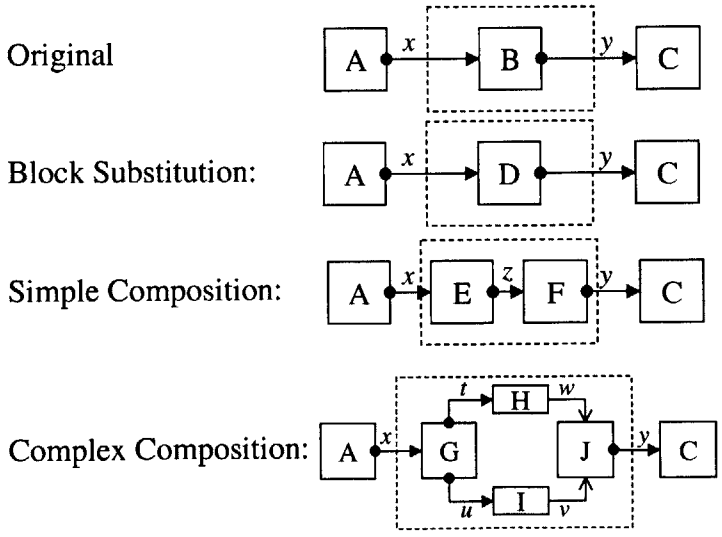


Figure 7.3 Simulation Flexibility. Original system shown (top) with some permissible substitutes.

initial three-module simulation, increasingly complex variations are presented. The simplest substitution is achieved by replacing a module directly with another. For instance, one controller may be replaced by a more sophisticated one. Slightly more complex is the substitution of a composed segment. Several modules now replace a single block, but the same input and output interfaces are maintained[1]. This process can be expanded to include more complicated substitutions as well as multiple information paths.

It is important to note exactly what is meant by flexibility. Each process is charged with maintaining certain communications links. GRRDE flexibility implies that certain module substitutions can be made without having to alter the logic within the remaining blocks. The communications services are abstract and the interfaces are transparent This only applies to the principles of information delivery. Reconfigured systems will be able to establish all specified communications paths but it is impossible for the GRRDE system alone to guarantee that the function of a composed segment will be as effective as the original. In fact, the whole reason to make substitutions is to change the behaviour of the entire simulation. It is the responsibility of the designer to account for processor usage, synchronization, and other impacts of the changes.

7.1.3 An Architecture Example

Architecture discussions without concrete examples are sometimes difficult to absorb. This section presents the information architecture of a simple controller simulation. It is crafted to highlight some of the commonly encountered design issues. A control example was judged to be the most intuitive illustration for these concepts, but the principles can be extended to other situations.

Consider a simple digital control loop. The controller receives an external set-point. Based on the commanded output and the current sensor reading, it must calculate an actuator command. The actuator block converts the abstract action into appropriate outputs based

1. It is permissible for the composed system to present a super-set of the original interfaces

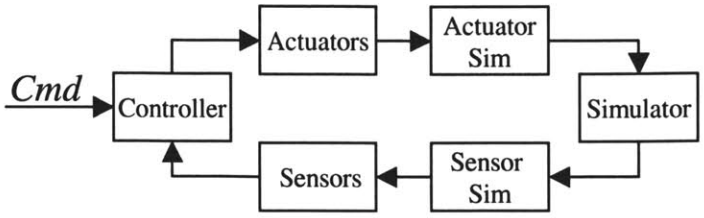


Figure 7.4 Conceptual design of a control simulation.

on the hardware represented in the simulation. This output is then fed into the actuator simulator which can add noise, bias or other non-idealities. The net actuation is then coupled to the dynamics simulator. This must add the effects of any external disturbances as well as propagate the dynamics forward in time. To complete the loop, the propagated state is filtered through the sensor simulation. After injecting appropriate non-idealities and synthesizing bit-level representations, the flight-software sensor block receives the data. This block may perform some filtering or pre-processing before passing the reading to the controller.

To understand the information architecture of this example, the figure can be redrawn (Figure 7.5) using GRRDE symbols. Command inputs are relayed to an internal process.

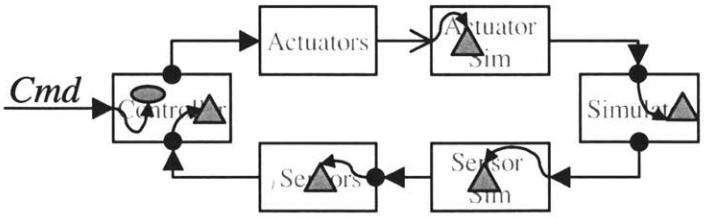


Figure 7.5 Communication elements displayed in control design.

This decision may reflect the intention that the controller will achieve each set point sequentially, or employ some manner of output shaping. Controller outputs are pushed to the actuator module. The relay function passes the commands to the simulator and makes the appropriate conversions to reflect the command structure of the hardware (e.g. torque scaled to voltage). The actuator simulator receives the input and records the value, while

the module logic adds noise to the desired command and calculates the net actuation. The dynamics simulator establishes a contract to obtain the actuator output (typically a force or torque) and uses the value in propagating the dynamics forward in time. This gives the 'true' value of system states such as speed or position. These values are then pushed to the sensor simulator at regular intervals. The pull contract from the sensor module is analogous to polling a hardware device.

An alternate architecture for the actuator simulator might involve a second relay from the actuator simulator to the dynamics simulator. The best choice of architecture may depend on the nature of the simulation. If control actions are sporadic (e.g occasional thruster firings) the relay response might be most efficient. For controllers that employ a continuous command application, the state input to the actuator simulator is preferred. If the sensor response is meant to be interrupt driven, this aperiodic behaviour can be mimicked through corresponding changes to the sensor branch.

The controller and the simulator are the key modules in the simulation. They are responsible for maintaining the bulk of the message pathways. These modules correspond to the primary functions of the simulation. The remaining modules maintain the connections across the simulator/FSW boundary. Particular system blocks may be replaced as the controller design matures (controller), the actuators change (actuator/actuator-simulator), more dynamics are modelled (simulator) or different sensors are employed (sensor/sensor-simulator).

The above discussion suggest common idioms for applying GRRDE services to embedded control systems and GFLOPS simulations. Our examples are not intended to be restrictive, they merely illustrate systems design concepts that we have found useful in our own development.

7.2 Real-Time Considerations

Software developed for hard real-time applications must typically be subject to rigorous analysis and a variety of tests to characterize its behaviour under a variety of operational conditions. Determinism, as always, is the primary concern and a number of issues contribute to this topic. First, designers must be satisfied that all tasks in the system will meet their individual deadlines. Second, asynchronous interactions between system processes can sometimes create a variety of undesirable conditions such as deadlock or priority inversion. Steps must be taken to address these risks. Third, although the execution of each GRRDE module is largely independent, efforts must be made to ensure that certain operations are synchronized or ordered.

7.2.1 Scheduling Analysis

One of the defining characteristics of a hard real-time system is the importance of task deadlines. Deadlines can be applied both to periodic and aperiodic tasks and specify the latest time at which the task may complete. When developing software for embedded systems, we must be able to guarantee that all tasks will meet their deadlines. This analysis should be performed during system integration and scheduling tests.

Priority Assignments

Modern real-time operating systems use dynamic, on-line process scheduling to avoid the inflexible nature of static scheduling. Generally, solving the scheduling (or bin-packing) problems is NP-hard [Burns & Wellings, 1996]. Thus, to reduce the complexity of on-line scheduling, sub-optimal algorithms are usually employed. A computationally simple approach is to issue each process a fixed, numerical priority and give control of the CPU to the enabled process with highest priority. OSE uses this type of on-line scheduling.

Rate Monotonic Priority Assignments (RMPA) [Liu & Layland, 1973] are ideal for periodic processes in the GFLOPS environment. Assigning priorities is necessarily a global

activity. All processes on the CPU must be considered together. This requires that we break the encapsulation of the individual modules and look at the processes inside.

Assessing Scheduability

Merely making priority assignments is not enough to completely assess whether process deadlines can be met. Designers must consider how the dispatch activities and overhead affect the system performance. We propose the following analysis framework.

Consider all periodic processes P_i, with period τ_i, and execution time C_i. In the absence of GRRDE we use the *Critical Zone Theorem (CZT)* [Liu & Layland, 1973] to check for scheduability. This test involves the hypothetical execution where all tasks are started at once. If every process meets its deadline in this execution, then any other set of start conditions will satisfy all deadlines. To assess the impact of GRRDE subscriptions we must take dispatch time into account.

To capture the effect of periodic subscriptions we propose augmenting the set of user processes with the hypothetical tasks Q_j. Since subscription periods are discrete the period of the task Q_j is just the subscription period τ_j. The execution time C_j will be found by the summation over all active subscriptions s with period τ_j:

$$C_j = \sum r_s \Big|_{\tau(s) = \tau_j} \tag{7.1}$$

The quantity, r_s, is the bound on the execution time of the dispatch function associated with subscription s. The CZT can then be applied directly to this new set of processes to determine scheduability.

Change-triggered subscriptions can also be accounted for in a straightforward manner. Sha, *et al* [Sha, et al, 1994], suggest augmenting the execution time of a process by its maximum blocking time B_i. Thus,

$$C_i' = C_i + B_i \tag{7.2}$$

The blocking time of a process can be found by considering the sum of the bounds to write to an *atomic2* object and read from any input object.

Unfortunately, most of these analysis techniques are most effective for periodic tasks. Accounting for the behaviour of aperiodic tasks involves one of the following:

- Assign aperiodic tasks priorities lower than all periodic tasks. This insures that periodic deadlines are met, but generally cannot guarantee much about the response to aperiodic tasks [Buttazzo, 1997]
- Model aperiodic task as periodic. If there is a minimum interarrival time, an aperiodic task could be considered periodic for the purposes of computing utilization. This can provide guaranteed deadlines, but may result in under-utilization of the processor [Kolcio, et al, 1999].
- Modify or augment the process scheduler. Many schemes have been proposed to provide better responsiveness to aperiodic tasks. Typically, they involve setting aside a portion of the execution cycle for aperiodic tasks. Some examples include: slack-stealing [Lehoczky & Ramos-Thule, 1992], aperiodic servers [Strosnider, et al, 1995], and reservations [Shin & Chang, 1995]. This is still an open field of research and there does not appear to be general consensus on the best approach.

Aperiodic task scheduling is still an active area of research. Users are encouraged to adopt a scheme that best matches their system characteristics.

Two issue are worth noting about this approach to accounting for GRRDE overhead. First, it requires a fairly mature system design. GRRDE allows flexible contract creation, but the *ad hoc* use of the publish-subscribe services may affect scheduability. Second, the analysis is most tractable when dealing with periodic processes. We have not investigated the application of this technique to aperiodic tasks, but the general approach should be compatible with some of the aperiodic analysis techniques described above. Furthermore, we conclude (and emphasize) that most of GRRDE's overhead comes from calling dispatch functions. The more efficiently they are written, the smaller the performance penalty.

7.2.2 Process Interactions

Distributed computing and preemptive multi-tasking create opportunities for unforeseen interactions between processes. When these interactions have a negative effect on system performance, their effects must be eliminated or managed. This section examines two common phenomena and the considerations built into GRRDE to help avoid them.

Priority Inversion

In the normal execution of a preemptive environment, if two processes are both enabled, the higher priority process should have control of the CPU. One potential hazard of mutual exclusion protocols (and semaphores), is the risk of the lower priority process preventing the execution of the high priority process for an arbitrary amount of time. This scenario is called *unbounded priority inversion* and is a common, harmful problem.

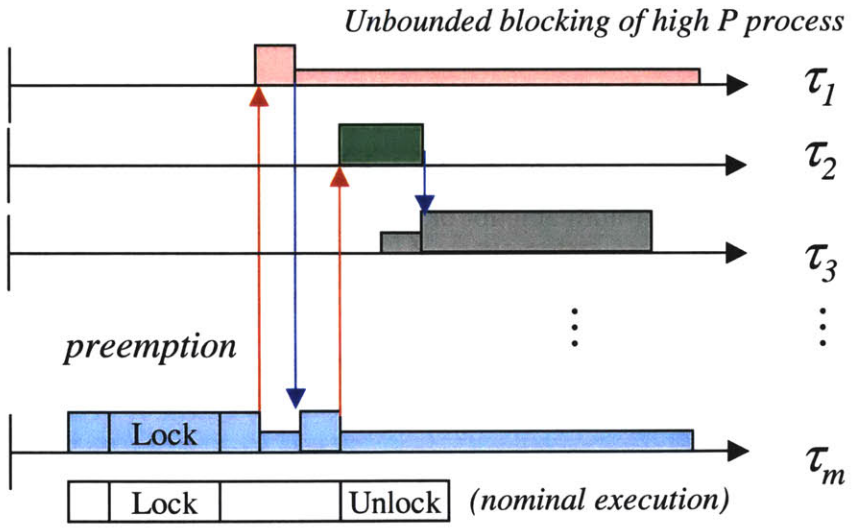


Figure 7.6 An illustration of unbounded priority inversion. τ_1 has highest priority.

Figure 7.6 shows a typical priority inversion scenario. We have a set of tasks τ_i, ordered in decreasing priority. Some system resource must be shared exclusively by processes τ_1 and τ_m. A process needing access to the resources atomically 'locks' a semaphore when

starting and 'unlocks' it when done. Processes attempting to lock an already secured semaphore will be blocked. In this scenario, τ_m becomes enabled and secures the semaphore. Before it finishes executing, process τ_1 becomes enabled, but blocks when it sees that the semaphore is not free. In the normal course of execution, τ_m would normally finish its calculations and release the semaphore allowing τ_1 to run. Trouble arises, however, when intermediate priority processes such as τ_2 and τ_3 become enabled and preempt process τ_m. As a result the high priority process τ_1 is prevented from executing for an unbounded amount of time by the execution of lower priority processes.

The main risk of priority inversion in GRRDE modules occurs when user processes interact with atomic objects that are used in published services. The high priority dispatcher or input arbiter could be blocked by a low priority user process during a read or write operation. To mitigate the effect of priority inversion, GRRDE atomic objects employ the *emulated priority ceiling* protocol [Sha, et al, 1994]. When a process attempts to lock a semaphore, it will temporarily raise its priority to that of the highest priority process sharing the atomic object. After unlocking the semaphore, the process restores its original priority. This prevents preemption by mid-priority processes and guarantees a maximum blocking time equal to $N_{sem} \cdot T_{ME}$, where N_{sem} is the number of processes sharing the resource, and T_{ME} is the maximum amount of time a process will need exclusive access.

In general, the use of protected memory spaces and modular design, reduces the reliance on mutual exclusion and restricts consideration to processes within a module.

Deadlock

When execution progress is halted due to unsatisfiable mutual dependencies, the system is said to be deadlocked. A simple example with two processes and two semaphores is shown in Figure 7.7. The processes must secure both semaphores at the same time in order to proceed with their calculations. Since they lock the semaphores in opposite orders, they have prevented each other from progressing. This cycle of dependencies is a common theme in identifying deadlock situations. Real world examples are often much more subtle

and may involve longer dependency cycles or race conditions [Burns & Wellings, 1996]. Other causes of deadlock include:

- Waiting for a stimulus from a deadlocked process.
- Message loss in a network environment.

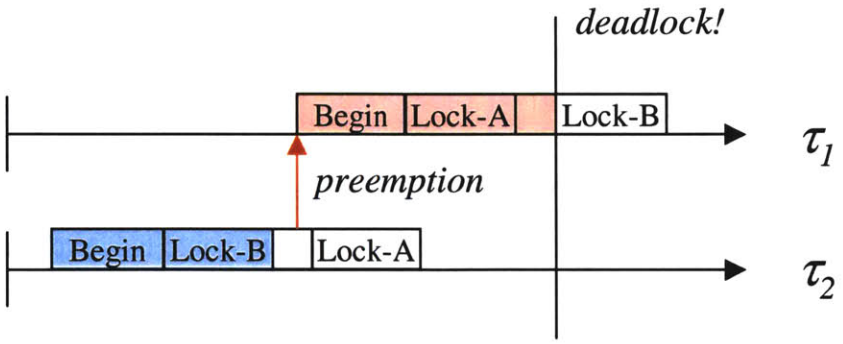


Figure 7.7 Deadlock between tasks accessing two semaphores. Since the tasks are waiting for each other, progress is impossible.

A number of algorithms exist that allow a distributed system to detect general deadlock situations. These strategies are useful, but require some insight into a class of interactions that may lead to deadlock. Qualitatively, certain design choices can lessen the threat of deadlock. For example, many blocking system calls like *receive*, have alternative forms that allow the user to specify a time-out period. Avoiding potential deadlock is an important part of embedded systems engineering.

GRRDE does not directly affect a system's risk of deadlock, but a number of design elements offer indirect benefit. First, localized dispatch mechanisms, help make GRRDE insensitive to network failures. Second, running a module's logic asynchronously with that module's inputs, allows the module to continue to execute, even in the absence of incoming data. This property must only be used after careful consideration since continued execution with stale data may be just as harmful as deadlock.

7.2.3 Synchronization and Consistency

Modern operating systems like OSE support flexible computational models. We have discussed earlier that simple cyclic executives are essentially one large loop with no true notion of processes concurrency (Section 2.1.2). More sophisticated systems may employ time-slicing to regulate a process's access to the CPU. Unfortunately, both of these systems are not very extensible, and provide mediocre response to aperiodic tasks [Locke, 1992]. OSE's event-driven, priority-based preemption scheme allows greater leeway in task execution, but is not without cost.

An OSE process can be enabled either by temporal or external stimuli. Message delivery can be used to explicitly direct the flow of execution, but we would like to preserve, where possible, the independent execution of each GRRDE module. Software system design will sometimes show that certain global ordering and data consistency properties must be maintained in order to ensure adequate simulation performance. This section examines some of these issues and ways of addressing them within the context of a GRRDE simulation. Each of these sub-sections describes common problems and outlines possible solutions.

Subscription Synchronization

Typically, cyclic tasks executing within a module will read data from a shared module variable, perform some processing on the contents, and then write the results to another variable. We have implemented a generalized type of atomic object that guarantees exclusive access to the variable during writing and reading. The contents of input variables are usually furnished by subscriptions, while output variables usually form the basis for the module's published services.

Consider Figure 7.8. The figure depicts two possible time traces of a module that performs calculations based on the values delivered in two separate subscriptions. In Figure 7.8a, the arrival times of the two subscriptions are widely spaced. Consequently, the values used in the module's calculations may be temporally inconsistent. Using quantities with poor

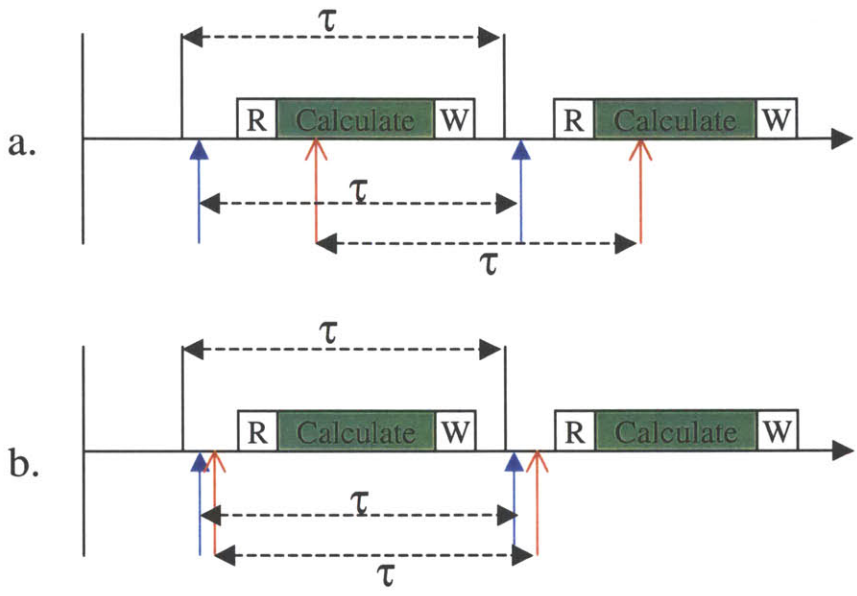


Figure 7.8 The importance of contract synchronization. In the top example (a), the timing of incoming messages (arrows) are poorly synchronized with the module's read-calculate-write cycle. In the lower figure (b), synchronization assures temporally consistent calculations.

temporal correspondence may result in degraded or faulty performance. In contrast, the message arrivals in Figure 7.8b, both occur before the calculation begins. Ensuring this behaviour requires two steps.

The first requirement is that incoming publish messages all arrive within a short time of one another. Designers can ensure that incoming subscriptions are aligned by specifying appropriate *first_fire* values when setting up contracts. This guarantees that the source values will arrive within a time window of width *pub_jitter*. Incidentally, 1 ms contracts will always be aligned under OSE since the operating system has a global time granularity of 1 ms. This feature is implementation specific and should not be relied upon.

The second synchronization consideration we must address is the execution of the module's calculation. Not only must incoming subscription messages arrive close together, but the module shouldn't start its calculations until the data are received. Several approaches can be used to address this problem:

- Trigger the logic process each cycle from the Input Arbiter. Once a new copy of each incoming publish message is received, the input arbiter generates a stimulus message and sends it to the logic process. This solution ensures good ordering every cycle but dealing with subscription failures or different arrival rates could be difficult. Care must be taken to ensure that the right "cluster" of sensor readings is used.
- Trigger the logic process once from the Input Arbiter. The logic process generates its own stimuli in steady state, but begins operations after being triggered from the input arbiter. If the cycle time for the process is sufficiently long (i.e. several ms or more), then we will have enough time-granularity to request a delay as large as *pub_jitter*. Thus, we will always maintain good temporal correspondence. During very short period cycles, we may not have the timing facilities to explicitly delay the execution of the Logic process until all inputs are received. In this case we must either adopt the previous triggering mechanism or tolerate the occasional delivery jitter.

Subscription synchronization can also be applied globally to ensure that sensor inputs are performed synchronously with actuator outputs.

Data Consistency

A similar problem that we may run into in a distributed, asynchronous setting is that of getting a 'snapshot' of the values of a set of atomic variables. This might occur when reading the input or output variables in a GRRDE module. Figure 7.9 shows a process P atomically reading the set of N, atomic variables X_i. Each variable is being written no faster than once every T_i time units. To obtain a snapshot we must read each value during an interval in which no writes occur. When we do not have control over the timing of these write messages we must resort to other means to obtain a consistent set of readings.

One possible solution is to implement a global mutual exclusion algorithm. This is a reasonable approach, but care must be taken to avoid unwanted blocking of other processes or potential dead-lock situations. If we can afford to tolerate some inefficiency, other solutions are possible.

Consider for example, the case where process P simply tries to read the values sequentially. If a new value is written before P reads X_N, then P must read each of the preceding

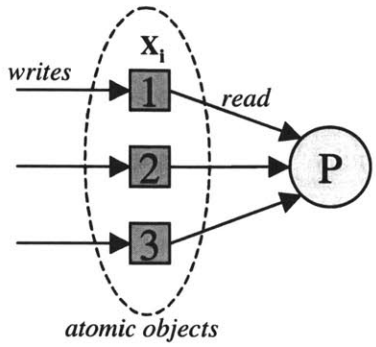


Figure 7.9 The Snapshot Problem. Process P must read the variables X_i during which no new values may arrive.

values once again. We assume for simplicity that read and write times both take time R_i. Table 7.1 shows worst case execution sequences for $N = 1, 2, 3$. We assume that the total time to read the values is less than the shortest period. Thus, the worst case will correspond to receiving a new write just before reading a new variable

TABLE 7.1 Worst-Case Executions for Snapshot Algorithm

N	Execution	Total Time
1	w_1 r_1	$2R_1$
2	w_1 r_1 w_2 r_2 r_1	$3R_1 + 2R_2$
3	w_1 r_1 w_2 r_2 r_1 w_3 r_3 r_2 r_1	$4R_1 + 3R_2 + 2R_3$

This pattern can be generalized yielding the time bound:

$$T_{max} = \sum_{i=1}^{N} (N + 2 - i) \cdot R_i \tag{7.3}$$

which is maximized if the R_i values are sorted in decreasing order. Thus, as long as $T_{max} < min(T_i)$, we are guaranteed to need no more than T_{max} time to complete. For a modest number of variables this approach may be a simple, self-contained way of obtaining a consistent snapshot. This algorithm is a variation of the bounded snapshot algorithm [Lynch, 1996].

Flight-Software/Simulator Synchronization

Successful GFLOPS simulations must synchronize the activities of both the flight software components and the simulators. Design of synchronization mechanisms between the two halves of a simulation is tricky. Not only must the actions of the various components be coordinated, but this must be done without skewing the flight-software to the demands of the simulation. This requirement suggests that the simulator be synchronized to the controller and not the other way around. Individual requirements may vary depending on the application, but the following general approach should be useful.

Consider the simple controller and simulator shown in Figure 7.10. The controller calculates actuator commands in response to the simulated inputs. A segment of the simulation execution is shown in Figure 7.11. We assume that the simulator period is τ_{sim}, and the controller period is τ_c. At the start of a simulation cycle t_0, the simulator begins propagating the system dynamics up to the start of the next cycle. This is the simulation horizon. This simulation process takes time t_{sim} to complete. Assuming that $t_{sim} < \tau_{sim}$, new data is available for publishing at time $t = t_0 + t_{sim}$. For the moment, we assume that $\tau_c = \tau_{sim}$.

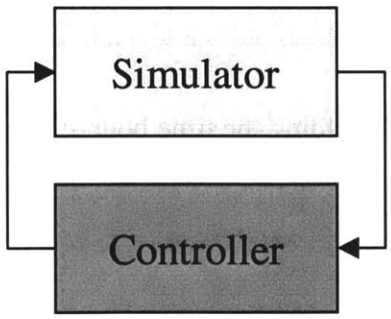


Figure 7.10 A very simple simulation.

We allow a timing offset of δ between the controller and the simulator. If

$$\tau_{sim} - t_{sim} < \delta \tag{7.4}$$

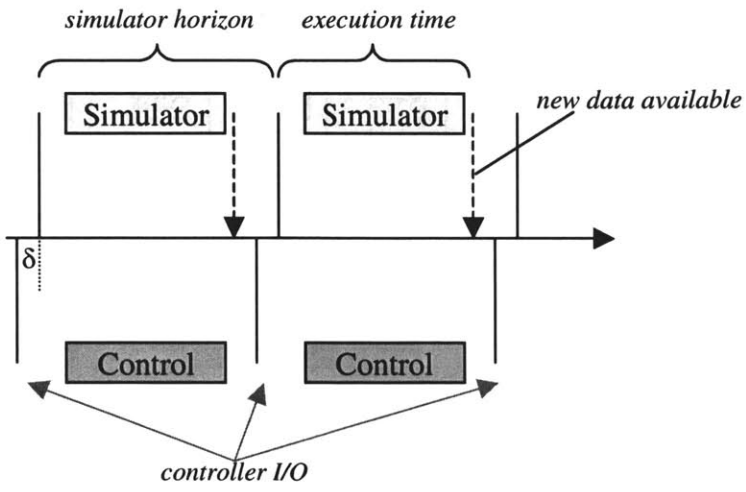


Figure 7.11 Timing diagram of simulation execution.

then the simulator and controller are essentially synchronized, and the sensor readings and actuator commands exhibit good temporal correspondence. In a more general case, we relax our initial assumption about the equivalence of τ_c and τ_{sim}. In Figure 7.12 we see a simulation in which the simulator operates at twice the rate of the controller[1]. In general, let us assume that:

$$\tau_c = R \cdot \tau_{sim} \tag{7.5}$$

for some integer rate multiplier, R. If we leave the definitions of τ_{sim} and t_{sim} unchanged, it is apparent that the condition specified in Eqn. 7.4 still holds. If t_{sim} is proportional to τ_{sim}, then the maximum allowable time mis-alignment, δ, will *decrease*. Therefore, increasing the simulation rate requires more accurate synchronization between simulator and flight software.

If we do violate this alignment bound, then the temporal consistency of our simulation will suffer. Essentially, we will have the situation shown in Figure 7.13. Because of the time

1. The (external) simulation horizon should not be confused with any (internal) simulation step size. The horizon is the granularity at which the user correlates simulation results with the actual passage of time. Internally, a simulation may use any necessary discrete or continuous propagation technique.

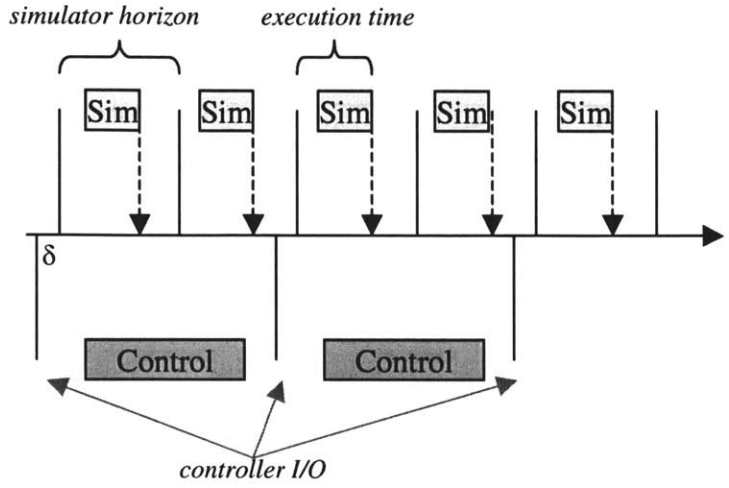


Figure 7.12 Timing diagram with mixed-rate simulation.

offset, the controller output at t_1, will only affect the simulator at t_2. When the controller next reads its sensors at t_4, it will only observe the propagation of the actuator commands up to t_3. Although the actuator commands from t_1 will be effective from t_2 to t_5, the

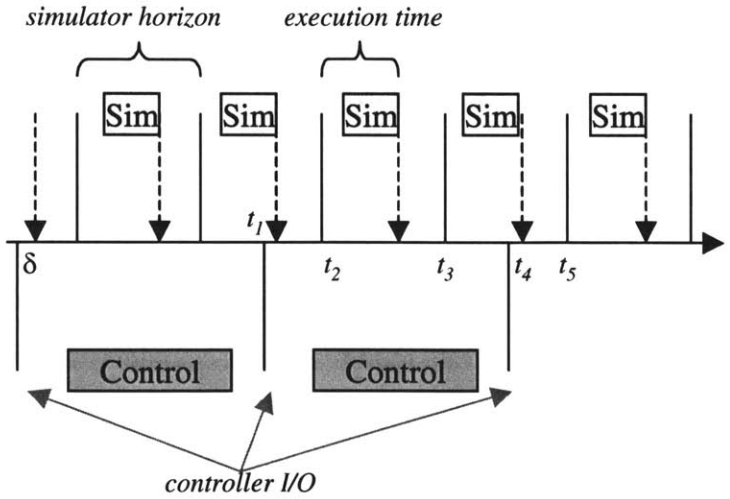


Figure 7.13 Timing diagram with time misalignment.

time misalignment introduces a source of unmodelled lag between sensing and actuation. The size of this lag is clearly:

$$\Delta = \frac{\tau_c}{R} \tag{7.6}$$

Such delays may significantly degrade the usefulness of the simulation results.

The above analysis suggests two approaches to simulation design. First, it may be possible to run the simulator fast enough (i.e. large R) so that the lag introduced in Eqn. 7.5 causes an imperceptible change in performance. In this case we will synchronize the controller and simulator whenever possible, but performance will be adequate even if precise temporal alignment cannot be maintained. If the flight software filters the incoming data the degradation from alignment lag may be reduced. This approach is simple but somewhat unpalatable since it introduces implicit assumptions into the overall simulation design. Alternately, if the simulation is run at the same rate as the controller (i.e. $R = 1$), we are afforded the largest margin of timing misalignment δ, but must endure the worst consequences if we violate the time-bounds. Deciding between the two options requires weighing the relative risks involved.

7.3 Deploying GRRDE Flight Software

The guiding architectural principle behind GRRDE is to provide an environment in which the transition from software development to software deployment is as painless and simple as possible. It is unlikely that any such activity will be completely effortless, and no amount of software simulation can eliminate the need for thorough testing. However, foresight and careful simulation may avoid most pitfalls.

To adhere to the principle of "fly as you test, test as you fly," GRRDE simulations can be constructed with an arbitrary level of flexibility. Simulators can reproduce, to bit-level accuracy, the input and output characteristics of sensors and actuators. Inevitably, there will always be the need for minor reconfiguration of these interfaces as a system moves to real hardware.

Our suggested migration changes are shown in Figure 7.14. The left side of the figure shows possible architectures for sensors. During simulation, the input values are obtained through a pull contract which puts the bit-level representation of the sensor readings into local storage. Any local processing will operate on this state, converting it to output variables. For deployment, the contract can be replaced by a periodic process which will poll the appropriate I/O device and copy the result into the same input state buffer. Actuators typically operate in a slightly different fashion. A relayed input is first processed to match the abstract input to the corresponding bit values. These are passed to a second process which sends the values in a signal to the actuator simulator. A deployed system must replace the second process with one that will write to the I/O device. In both cases the amount of code that must be altered remains small. Actual implementation may differ based on the application, but this example illustrates the general principles involved.

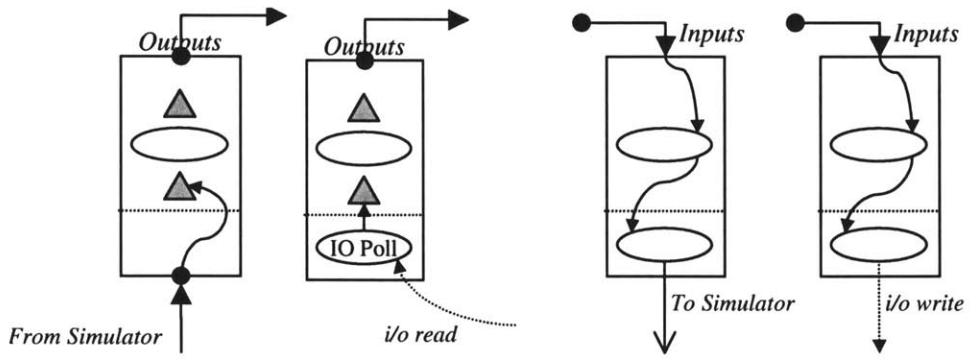


Figure 7.14 Migration of interface software. Sensors (left) and Actuators (right) require slightly different strategies.Features above the fine dotted line remain intact through transition.

7.4 Summary

We have attempted, in this chapter, to give an overview of large-scale, real-time, simulation design using GRRDE and GFLOPS. We first examined some of the architectural issues involved in structuring the module connections. We also examined the documentation required at the module interfaces, especially when specifying available subscriptions.

Next, we examined the way in which GRRDE would fit into a conventional real-time analysis and testing program. Scheduling and synchronization were of particular concern. Finally, we sketched out an approach to migrating simulations and flight software to hardware testbeds and deployed systems.

This chapter concludes our documentation of the GRRDE middleware. The preceding four chapters present a thorough discussion of design and application of the publish-subscribe services. To further emphasize the usefulness of our software development framework, the next sequence of chapters presents several case-studies showing the usefulness of GRRDE for integrating advanced software concepts into conventional flight software.

Chapter 8

AUTOMATIC CODE GENERATION AND REAL-TIME WORKSHOP

Abstraction is a principal feature of good systems design. Separating the details of implementation from its functional representation allows domain experts to concentrate their design effort where it is most productive. We have presented the GRRDE system as a communications abstraction tool that reduces the complexity of coordinating software module connectivity in distributed systems. Other abstraction methods are also common in systems design.

One common innovation in the development of embedded control systems is the use of automatic code generation. These techniques and tools allow direct generation of real-time software from block-diagram models of monitor and control systems. This chapter examines the integration of GRRDE middleware with a common code generation package. We use the Real-Time Workshop (RTW) and Target Language Compiler (TLC) components of Mathworks's Simulink software to directly generate GRRDE-compatible software modules from their Simulink representations. This method is extremely effective and can be used to rapidly build libraries of interoperable flight software modules or simulations.

8.1 Background

This section provides the motivation and background information for this application study. We first provide a brief introduction to the practise of automatic code generation

(ACG), and then explain the role of our study with respect to the evolution of GRRDE middleware.

8.1.1 Why Code Generation?

Automatic code generation may seem like a strange and revolutionary approach, but it is really just a logical extension of the evolution of modern computer languages. These tools essentially constitute a high level, domain-specific computer language. Often graphical in nature, they allow control engineers to program embedded devices in an intuitive manner. This provides an array of benefits to product development. Already popular in the automotive industry, ACG has also been used in aeronautical and space applications and is steadily gaining acceptance.

Software engineering differentiates between low-level languages like assembly language, and high-level languages like C, Ada, or Java. The terms 'high' and 'low', essentially refer to how abstract the language features are. Languages with vocabulary and structure closely tied to simple machine operations are considered lower-level than those languages with more general features. If the primitives of the language match the task you are trying to accomplish, development will be much easier. For example, developing I/O device drivers is more easily done in C than in LISP; expert systems are just the opposite. ACG tools allow control system designers to model controllers and simulations out of conventional elements such as filters and integrators. Construction is performed graphically (Figure 8.1) and involves connecting data pathways as if they were wires. The completed design is compiled by the tool into a standard language such as C or Ada, and from there, into assembly language.

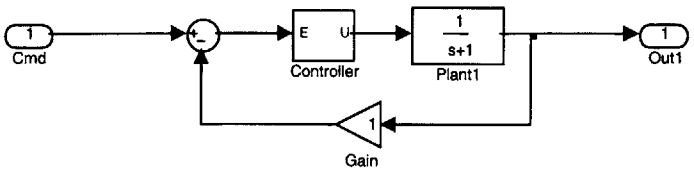


Figure 8.1 A very simple control system model in Simulink.

The benefits of ACG for product development are manifold.

First, code generation reduces the cost of development [Smith & Elbs, 1999] and the design iteration time [Orehek & Robl, 2001]. Using traditional, hand-coding techniques, control engineers design the control algorithms and derive specifications. These specifications are given to the software engineers, who implement and debug the code. The design is tested against simulations or hardware and the results returned to the control engineers. Revising the design is a time-consuming process. Using ACG tools, the control engineer can create prototype code directly for testing. Although the final production code may be translated or optimized by hand [Maclay, 2000], the development phase is substantially accelerated.

The second benefit is increased safety and reliability. Since the design is formulated by control engineers, but implemented by software engineers, errors may be introduced at the interface between these two groups. Without well managed communication, misunderstandings are frequent. Changes that appear unimportant to the software engineer may be critical to the controller's performance. Moreover, several secondary safety benefits accrue. When design iterations take significant effort, designers will frequently tweak the code by hand between revisions. This leads to divergence between the specification and implementation. Shorter iteration time significantly reduces this temptation. Furthermore, when projects require lower volumes of hand-crafted code, software engineers are more likely to start from scratch and not reuse inappropriate old code.

These benefits of code generation tools have led to growing popularity in many areas of embedded systems development. Automotive applications abound [Orehek & Robl, 2001][Maclay, 2000], since cars are increasingly reliant on software. Engine and emissions control, braking, steering, and cruise control are all coordinated by microcontrollers. Many aircraft flight control systems have been reliant on software for a while, but are adopting ACG to reduce costs and improve reliability [Bryant & Key, 1999]. Space applications are also becoming popular. ACG was used to create flight software for the Delta

Clipper experimental rocket [Nordwall, 1993], and to prototype flight instruments [Ptak & Foundy, 1998]. Many more examples can be found in scientific literature, and international conferences have been devoted to this specific topic.

8.1.2 Study Goals

Popularity aside, we are left to explain how ACG ties into our middleware development. Our central goal is to integrate code generation tools with GRRDE. This study extends the Simulink/Real-Time Workshop utilities so that designers may directly translate their simulink modules into GRRDE-aware modules. Specifically, these modules will publish their own outputs and set up their own subscriptions. This suggests the following benefits:

- Accelerates iteration cycle. This aids the control system developer by reducing the effort required to integrate ACG modules with other system components.
- Permits library-based development. Since the interconnection mechanism between modules is abstract, simulator and controller modules can be quickly assembled or reconfigured.
- Promotes reliability. Communications abstraction reduces reliance on hand-coding module interconnections.

Providing these benefits to the control engineer demonstrates that GRRDE helps to support control system development. This utility is of value to flight software in general.

8.2 Code Generation and GRRDE

Having decided to integrate GRRDE with automatic code generation tools, we now examine how this was accomplished. The ACG tool chosen for this study was the Real-Time Workshop (RTW) module of Simulink. Simulink is a popular block-diagram, control-systems design tool. It is part of MathWorks's MatLab. RTW was selected for this study primarily for its ready availability, and familiar simulink interface. Other tools, such as Matrix_X are quite popular and could also be adapted in this manner. Let us examine how the typical RTW build process operates, and how it can be adopted to work with the GRRDE middleware framework.

8.2.1 Real-Time Workshop

To understand how we adapted RTW to interface with GRRDE, we must first examine the nominal build process. Compiling executable software with RTW takes several steps (Figure 8.2). First, the developer creates a standard simulink model using the graphical development tools. Global parameters such as time-step or integration method can be specified in this stage. When the RTW engine is invoked, the users's model is converted from a conventional '`.mdl`' file into an '`.rtw`' file. This file is a pre-compiled version of the users's model. It contains all the required information to replicate the function of the simulink model, without extraneous information such as graphics settings. The rtw-file is passed to the *Target Language Compiler*, which translates the model into functional code. Users have the option of targeting different languages, architectures and operating systems. The final stage is performed by a conventional compiler (or cross-compiler). The

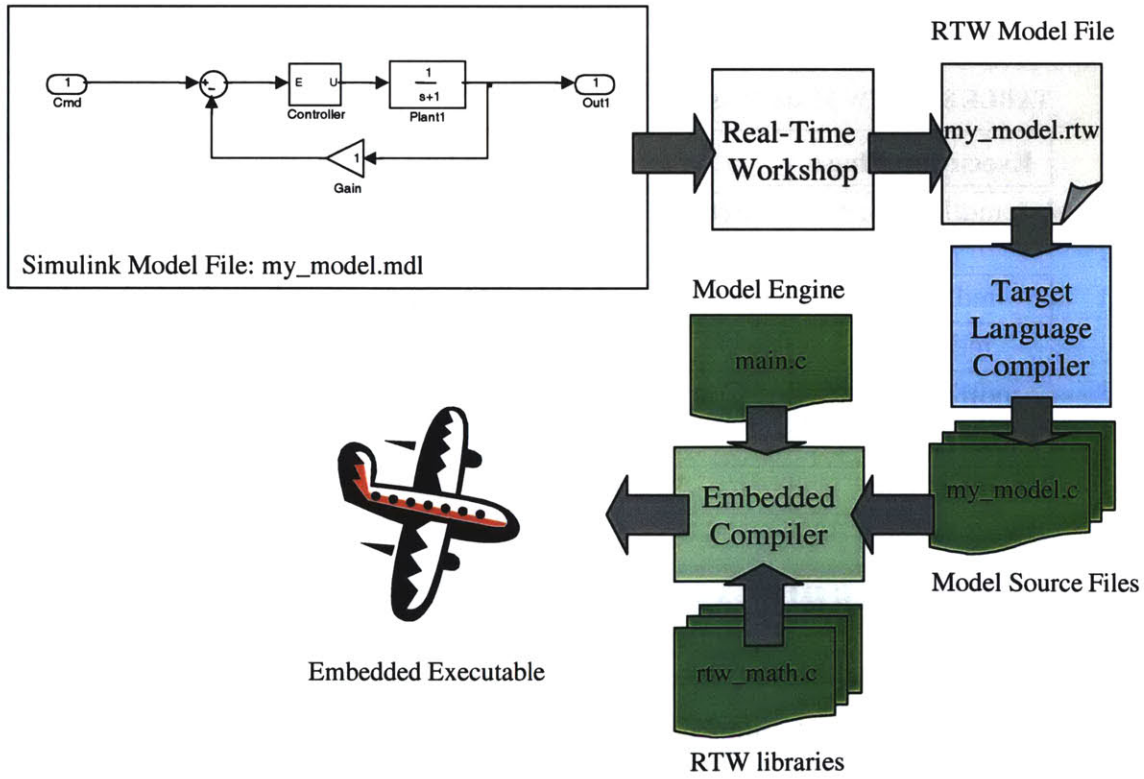


Figure 8.2 Real-Time Workshop Compilation Sequence.

compiler takes the model source files and combines them with libraries of specialized math routines and a model engine. The math routines include support for operations such as filtering, FFTs, integration, etc. The model engine is an application skeleton that is designed specifically for the targeted operating system or embedded processor. It is responsible for interfacing with the native timing routines and periodically executing the model.

RTW is able to support a wide array of embedded targets by adopting a common application framework. Model execution is divided into a number of stages. The key stages are shown in Table 8.1. Each stage is implemented as a predefined function with a standardized parameter list. These functions are called at appropriate times during the periodic operation of the model engine, but the contents of the functions are generated from the model source files (generated by the TLC). Every block in the simulink model contributes to one or more of these functions. Thus, a single version of the executable engine can be used for any user model.

TABLE 8.1 RTW Model Phases

Execution Phase	Purpose
mdlInitialize	Sets up initial values in components and variables
mdlOutputs	Compute and propagate outputs at each timestep
mdlUpdate	Update discrete states in model
mdlDerivatives	Compute derivatives of continuous functions
mdlTerminate	Clean up after completion
mdlStart	Run once at beginning of execution

8.2.2 Developing GRRDE-Aware Modules

The code generating facilities of Real-Time Workshop allow extensive user customization. Users may select from a number of destination languages such as C or ADA. They may also select different target processing architectures and operating systems. Most importantly, users also have the ability to completely change the build process. New oper-

ating systems can be defined, and custom blocks can be created. This extensibility is what enabled GRRDE integration.

The modified development sequence is shown in Figure 8.3. We first create our simulation in Simulink. The parts of the model destined for code generation are placed in a sub-system and connected to the rest of the simulation with input and output ports. Once we are satisfied that the controller is ready for compilation, we replace the default I/O blocks with special GRRDE-enabled versions. When we invoke the RTW component to generate the real-time code, special template files translate these modules into source-code.

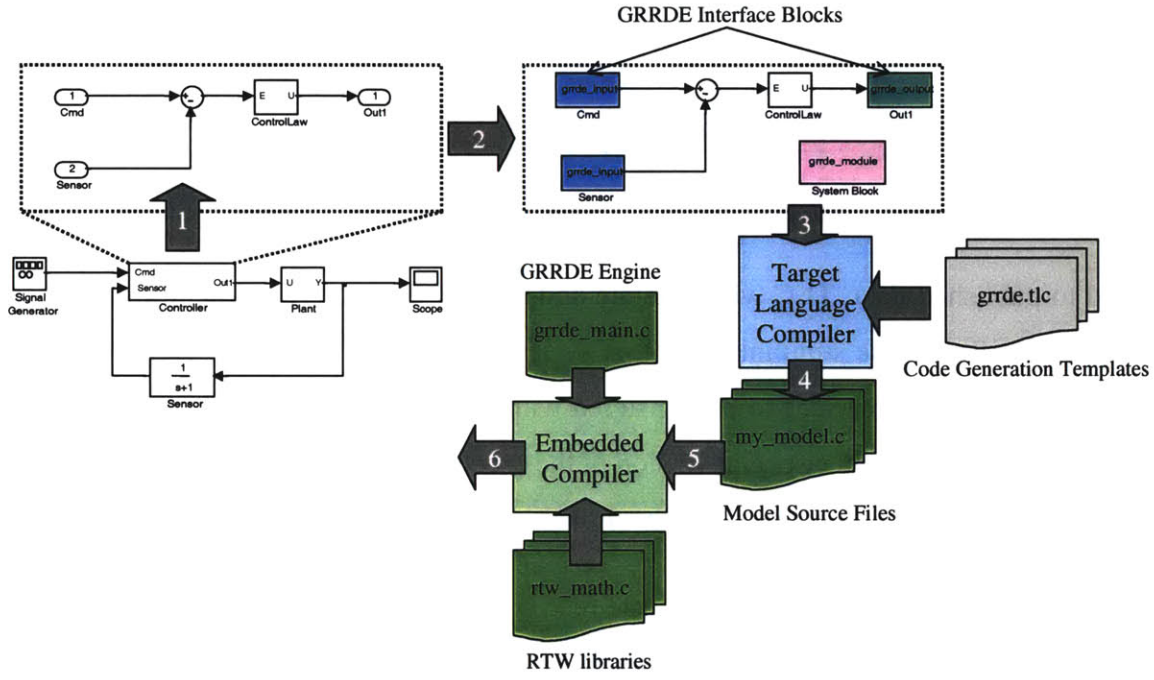


Figure 8.3 GRRDE custom code generation process.

Template '`.tlc`' files tell the TLC how to generate source code for our customized components. Input blocks correspond to subscriptions that must be requested. A custom input arbiter is created from the templates and entries are added for each type of input signal. The specifications for the subscriptions are set by changing the block parameters. A screen-shot of the input parameter dialog-box is shown in Figure 8.4. Output blocks corre-

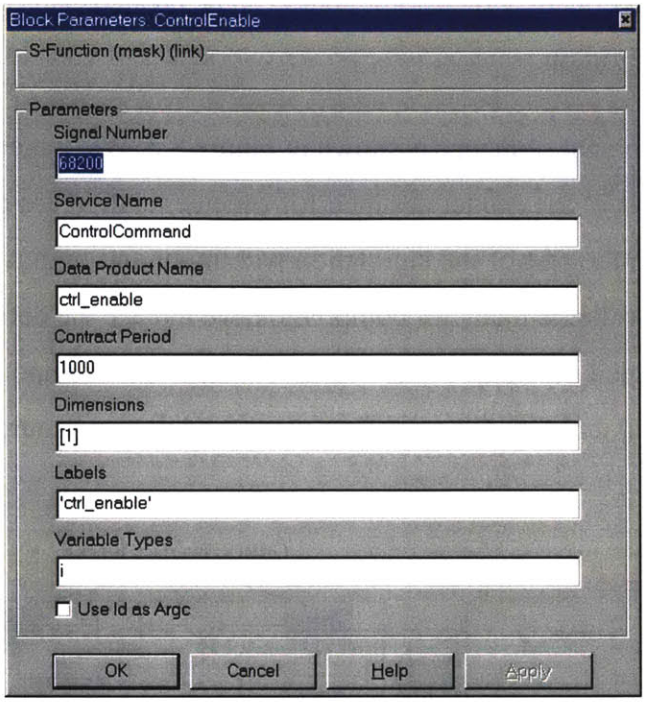


Figure 8.4 Example of grrde_input parameter-setup dialog box.

spond to published data. TLC will generate and register dispatch functions for each data product specified in the simulink model. A system-level block identifies the name of the module and the data services to be registered with the OSE-NameServer (Section 3.2.2). During final compilation, a special version of the execution engine interacts with OSE and creates the processes necessary to run the model.

This approach can be used to rapidly create both controller (flight-software) and simulator modules for use with GFLOPS and GRRDE. Modules can interact with other ACG software or conventional hand-coded GRRDE software. If necessary, the module source code produced by TLC can be easily edited by hand. This process is best understood by an example.

8.3 An Control System Example

In this section we present a very simple control system and compare the output generated through internal simulink simulation with the module outputs from generated GRRDE modules. The example we have chosen is an idealized attitude control system for a free-floating spacecraft. A simple pictorial representation of the system is shown in Figure 8.5. The control system can measure sun-angle with the sun-sensor and wheel-speed with a tachometer on the reaction wheel. The reaction wheel is the only actuator, and is commanded with torque.

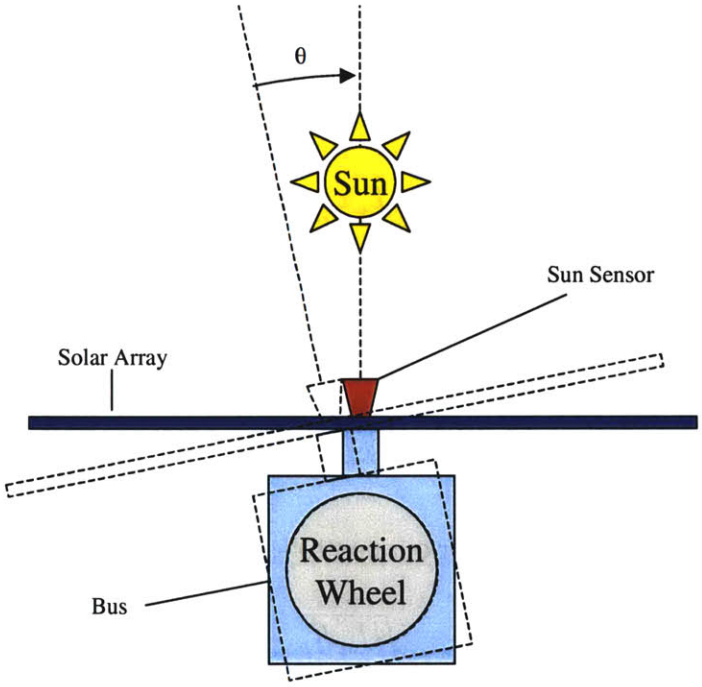


Figure 8.5 Single axis spacecraft attitude control. Sun sensor measures angle to sun (in spacecraft reference frame), wheel tachometer measures wheel-speed.

The dynamics of this system are fairly easy to describe. The satellite bus is modelled as a rigid body with moment of inertia I_{sc}. The reaction wheel has zero bias momentum and moment of inertia, I_w. The controller can command a torque, τ_c, to be applied to the wheel. The sun angle, θ, is measured in the frame of the spacecraft. Assuming right-handed sign conventions, a positive (counter-clockwise) rotation of the spacecraft yields a

negative sun angle. The angular velocity of the bus and wheel are ω_{sc} and ω_w, respectively. Thus:

$$\omega_{sc} = \omega_{sc_0} - I_{sc}\int \tau_c dt \tag{8.1}$$

and,

$$\theta = \theta_0 - \int \omega_{sc} dt \tag{8.2}$$

For our controller, we assume a very simple proportional-derivative (PD) control. Thus, our control law is:

$$\tau_c = -K_p(\theta - \theta_c) + K_d(\omega_{sc} - \omega_{sc_c}) \tag{8.3}$$

Since,

$$\omega_{sc} = -\frac{I_w}{I_{sc}} \cdot \omega_w \tag{8.4}$$

we can write:

$$\tau_c = -K_p(\theta - \theta_c) - K_d \cdot \frac{I_w}{I_{sc}} \cdot (\omega_{sc} - \omega_{sc_c}) \tag{8.5}$$

Using a spring-mass-damper model for our controller, critical damping occurs when:

$$K_d = \sqrt{4 I_{sc} K_p} \tag{8.6}$$

The simulink block diagrams for the controller and simulator are shown in Figure 8.6 and Figure 8.7. The simulator adds Gaussian noise to both the actuator command and the sensor readings. Prior to generating the real-time GRRDE code, we replace each of the I/O ports in these diagrams with a GRRDE-interface block.

Once the conversion is complete we generate the GRRDE modules. Figure 8.8 shows comparative results from a simple simulation ($\theta_0 = 1, I_{sc} = 100, I_w = 1, K_p = 1$). At

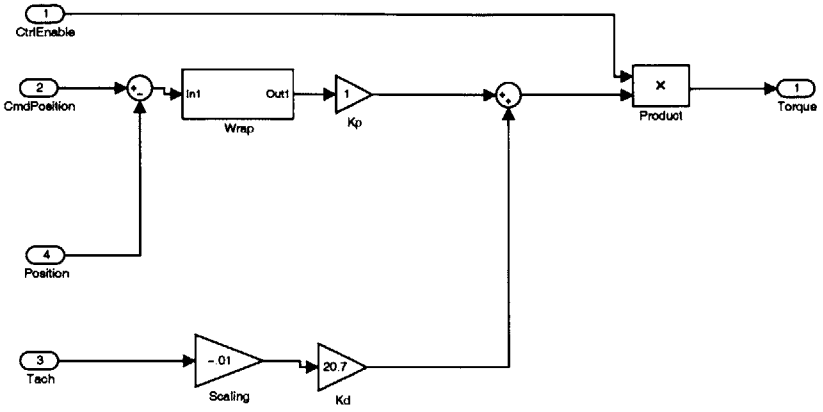


Figure 8.6 Controller Block Diagram.

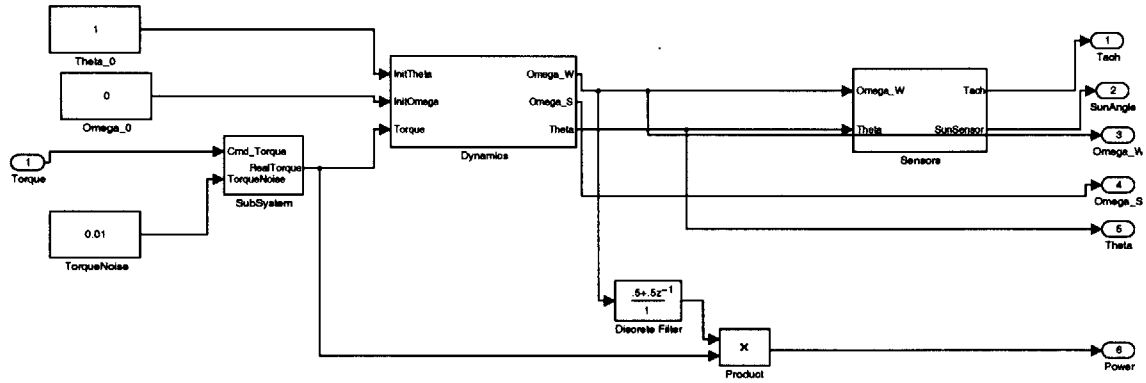


Figure 8.7 Simulator Block Diagram.

$t = 2s$, the controller is commanded to slew to $\theta = 0$. The two curves match very closely. The small deviations observed are caused by synchronization offset between the simulator and the controller.

Several similar examples were developed during our development of the RTW extensions and template files necessary to convert Simulink models to GRRDE modules. Since our goal was to establish the practicality of this technique, rather than perform a particular control system design we did not perform a comprehensive evaluation of the generated modules. Further testing and refinement of our RTW extensions would be necessary before using them in a demanding application.

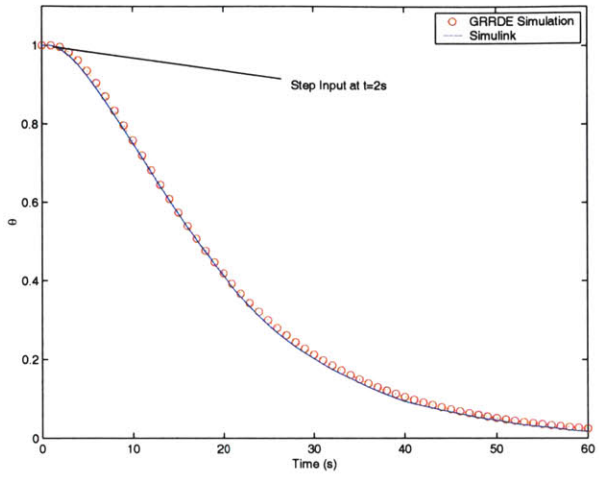


Figure 8.8 Step command response (Internal and GRRDE simulations).

8.4 Summary

We have demonstrated successful integration of the automatic code generation capabilities of Real-Time Workshop with the middleware functions of GRRDE. The following general observations were made during this study:

- Although RTW and the TLC perform most of the code generation tasks, the designer must still plan the module interconnections. They must provide specifications for signal identifiers, data structures, etc.
- The TLC templates are useful for creating skeleton modules, even if the logic will be implemented by hand. The automated generation of input contracts, the input arbiter, and the dispatch functions and module setup is very useful.
- The graphical design significantly improved the speed of our internal simulator development. Although we were unable to quantitatively measure this improvement in productivity, it was qualitatively significant.

Our RTW study was fairly short. Time did not permit the full range of testing and development. To extend the usefulness of our tool customization, we would make the following enhancements:

- RTW modules make no attempt to synchronize contracts or block execution. Presently, this must be added to the source-code by hand. It should be feasi-

ble to create custom simulink blocks that give the designer access to synchronization primitives.

- Periodic contracts are the primary means of module I/O. Command inputs were implemented by creating a GRRDE-Input module as if it were a subscription input, and then disabling the contract initialization in the source code. The block would recognize and use the input, but would not subscribe to it. It would be a straightforward manner to add direct support for this behaviour.
- Implementation of change-triggered publication services would also be useful. This would allow us to integrate GRRDE with the StateFlow toolbox (StateFlow adds state-chart control supervision to Simulink models).
- Simulation of spacecraft clusters entails running several parallel simulations. It would be useful to permit automatic replication of the simulators. Presently, the designer must either build a large model with all logic duplicated for each spacecraft, or load several copies of the simulator module at the same time.
- We have not extended the formal analysis or temporal characterization to include automatically generated modules.
- Most of the simulink models used for this study are quite simple. An expanded study with more challenging examples may provide further evidence of the usefulness of this approach.

In general, the results from this study are quite promising and suggest that middleware and code generation are complementary techniques. Both provide domain-specific abstractions and streamline development by keeping the designer's focus on relevant tasks.

Chapter 9

AUTONOMOUS FAULT DIAGNOSIS WITH GRRDE AND MARPLE

Spacecraft autonomy is currently one of the hottest topics in space software research. Successful, high-profile missions such as Mars Pathfinder have provided the public with visions of intelligent robotic explorers. Although somewhat less glamorous, but technologically more momentous, the Deep Space-1 technology demonstrator mission actually tested autonomous navigation, planning, fault diagnosis and execution in space [Rayman, et al, 1999]. Some designers have even gone so far as to propose building space probes *without any radio receiver* so that the on-board intelligence will conduct the entire mission. Few would suggest that current technologies are ready for this challenge, but it is an intriguing prospect nonetheless.

This chapter expands the role of GRRDE as a technology enabler. In this study we use GRRDE to integrate an autonomous fault diagnosis engine into a simple simulation. The flexible communications abstractions allow the new functions of the diagnostic engine to be layered on top of conventional flight software. GRRDE assists this integration in two ways. First, the abstract services allow us to tap directly into the telemetry stream. Second, real-time data delivery guarantees, reduce the potential for state confusion due to temporal inconsistency.

9.1 Motivation

The general aim of this study is to demonstrate how the GRRDE middleware can be used to support the integration of autonomy with conventional flight software. Autonomy is a rather loosely defined term and has been used to describe a wide range of behaviours. High level feedback control, navigation, maneuver planning, and self-repair are examples of the types of functions commonly found under the label of autonomy. This study focuses on fault diagnosis. In this section we provide a brief overview of autonomous diagnosis and an outline of our study goals

9.1.1 Fault Diagnosis

The primary purpose of a diagnostic tool is to estimate a system's state of health. We might be studying a person, a car's engine, or a spacecraft half-way across the solar system. Based on behavioural observations we must determine if the device is operating normally, and if not, what is wrong with it. Autonomous diagnosis is distinct from interactive diagnosis due to the limited observability of system state. A doctor can order blood tests, an electronics technician can wield a multi-meter but a satellite operator only has access to whatever self-instrumentation is built into the spacecraft. This section considers the benefits of autonomous diagnosis systems and some typical techniques.

Spacecraft, even cheap spacecraft, cost millions of dollars. When failures disable a satellite, fast, effective recovery reduces the potential for lost revenue or science. Troubleshooting anomalous behaviours in a remote spacecraft can be difficult, especially if communications have been disrupted. Fault diagnosis allows the spacecraft to identify its own failed components. This process is independent of the distance to Earth and can use any local information, not just the quantities that fit into the telemetry downlink. Some systems even have automated fault-recovery capabilities. Using these techniques allows the satellite to inform ground operators of faulty components or automatically activate redundant sub-systems during time-critical operations.

Most modern techniques for fault diagnosis are termed 'model-based'. Each component in the monitored system is associated with an operational model. These models provide simple sets of predictive rules that capture essential device properties. These relations can either be qualitative [Davis, 1984] (e.g. if the valve is *open*, and the tank pressure is *high*, then we should observe *forward thrust*), or quantitative [de Kleer & Brown, 1992] (e.g. the electrical power consumed by the reaction wheel is given by $P = (\tau \cdot \omega)/\eta$). Both approaches compare predicted and observed behaviours for deviations. Model-based approaches can usually tolerate more behavioural uncertainty than other methods such as rule-based systems [Sary & Werking 1997]. The method selected for our study is a quantitative diagnosis package called Marple [Fesq, 1993].

9.1.2 Study Goals

We have selected model-based fault diagnosis as our sample autonomy technique. This was judged to be a good example for a number of reasons. First, fault diagnosis has applications to practically any space missions from planetary probes to geostationary satellites. Second, since diagnosis tools can essentially run passively (i.e. look but don't touch), the risks associated with adopting them into a mission are more easily managed. This may improve their rate of adoption. Third, fault diagnosis is quite easily compartmentalized and can be employed on a fairly small scale. This permits us to define a small, well-bounded study.

Implementing autonomous capabilities in a GRRDE-based simulation helps to demonstrate the utility of the middleware. Our services make it possible to layer (Section 7.1.2) advanced functionality, like diagnosis, atop existing software. Expanded functional additions can represent the migration of capabilities from ground to space, or simply capability upgrades. GRRDE avoids software integration complexity and streamlines the development process.

9.2 The MARPLE Model-Based Fault Protection System

Marple is a quantitative diagnostic engine capable of handling significant noise and ambiguity in the modelled behaviour. Marple application begins during development, with the derivation of subsystem models. During operation of the system, observed sensor and actuator data are propagated through these models and the results are examined. If significant discrepancies are observed between observed and predicted values, Marple concludes that a fault has occurred and attempts to identify the faulty component. Once the diagnostic phase has begun, Marple generates a list of suspected components and sensors that may be responsible for the fault. If component reliability data are available, these results can be ranked in order of likelihood.

Developing component models first requires identifying input and output nodes for each component. Figure 9.1 shows a very simple model of an adder. There are two inputs (X and Y) and one output (Z). Nodes can be connected to *sensors*, to other component nodes, or to both. From a modelling perspective, Marple considers any observable input or output (I/O) value to be a sensor. Thus, a voltage reading is a sensor, but so is a torque command. Once the nodes, sensors and components have been identified, numerical constraints must be derived to relate node values. Logically, the forward constraint in the adder example is the $Z = X + Y$. We can also permute this equation to provide relations for the other nodes, i.e. $X = Z = Y$ and $Y = Z - X$..

Diagnosing faults consists of two steps. The first step is identifying that a fault has occurred. During operation of the modelled device, Marple obtains periodic readings from the sensors. Once a set of sensor readings has been received, Marple checks the observed values against those predicted by model constraints (Figure 9.2).The engine starts at each sensor and propagates the value through the models. Components with multiple nodes and constraints can create several propagated values. A value's propagation stops when there are no constraints to propagate or the value reaches another sensor. Once all sensor values have been propagated, the values at each node are compared to one another. Since we are

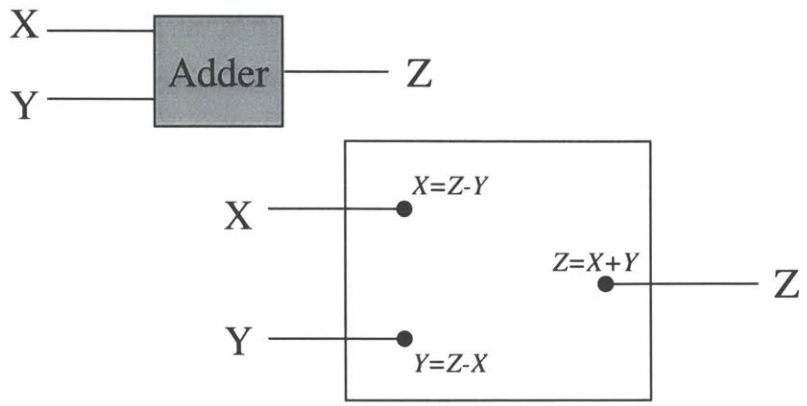


Figure 9.1 A simple Marple model. The modelling process consists of identifying components and writing constraints between each node.

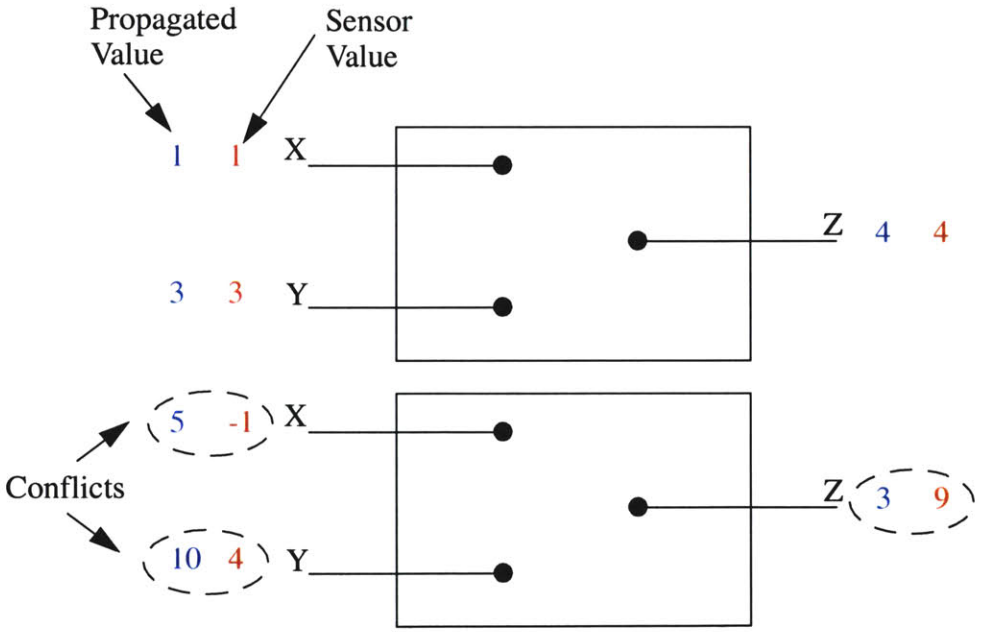


Figure 9.2 Constraint propagation for the adder. In the upper figure, the model is operating correctly, in the lower example the propagated values do not match the observed values indicating a fault in the system.

dealing with mixed analog and digital systems, the values are not compared for exact equality. Instead, node values within the absolute or relative tolerances specified at a node are considered consistent.

If these values do not match, the Marple engine registers that there is a fault in the system. In this, the second stage of the diagnostic process, Marple attempts to isolate the faulty component or sensor. Using a technique called constraint suspension [Davis, 1984], the engine will choose one component or sensor at a time and remove (*suspend*) its predictions from the system. The engine then checks the consistency of the modified model. If the removal of this component eliminates all the conflicts in the propagated values, then it may be at fault. These components are placed in the list of candidate diagnoses. The engine will suspend both components and sensors in its attempt to explain the faults in the system.

For complex systems, effective isolation of single faulty components, relies on having sufficient redundancy in the sensor array. In our simple example, the conflicts observed could be explained by a fault in any of the three sensors, or the adder itself. Although we know there is a fault the system is unable to isolate it. Thus, the overall system design can affect the usefulness of the diagnosis.

9.3 Integrating Marple with GRRDE Middleware

The developments in this study were aided by other research conducted at the Jet Propulsion Laboratory by current MIT personnel. Previous work included the implementation of the Marple engine in C++ and the development of a tool which allows users to create component modules using MathWorks's Simulink[1]. In this study we reengineered the shell-based Marple engine for the embedded GRRDE environment, modelled a simple scenario and tested the resulting system in real-time simulation.

9.3.1 Scenario Modelling

We have based our simulation on the simple attitude control system developed in Section 8.3. The basic flight-software is just the simple controller we developed previ-

1. Marple model creation uses a customized blockset; it does not automatically generate models from on a functional Simulink model.

ously. The simulator consists of two software modules. In addition to the basic dynamics simulator we also developed a secondary simulation module to represent the electrical subsystem. This module added inputs from two sensors. The first, is a voltage sensor to measure the output of the photo-voltaic array (PVA). It is sensitive to the degree of solar illumination. The second sensor is a binary battery discharge indicator. If the *true* value of the PVA voltage drops below a certain threshold V_T, the indicator will register the value *true*.

A block diagram of the electrical simulator is shown in Figure 9.3. The dynamics simulator was essentially unchanged from Figure 8.7 and includes a power-draw indicator for the reaction wheel. The simulators were created in Simulink and translated into GRRDE mod-

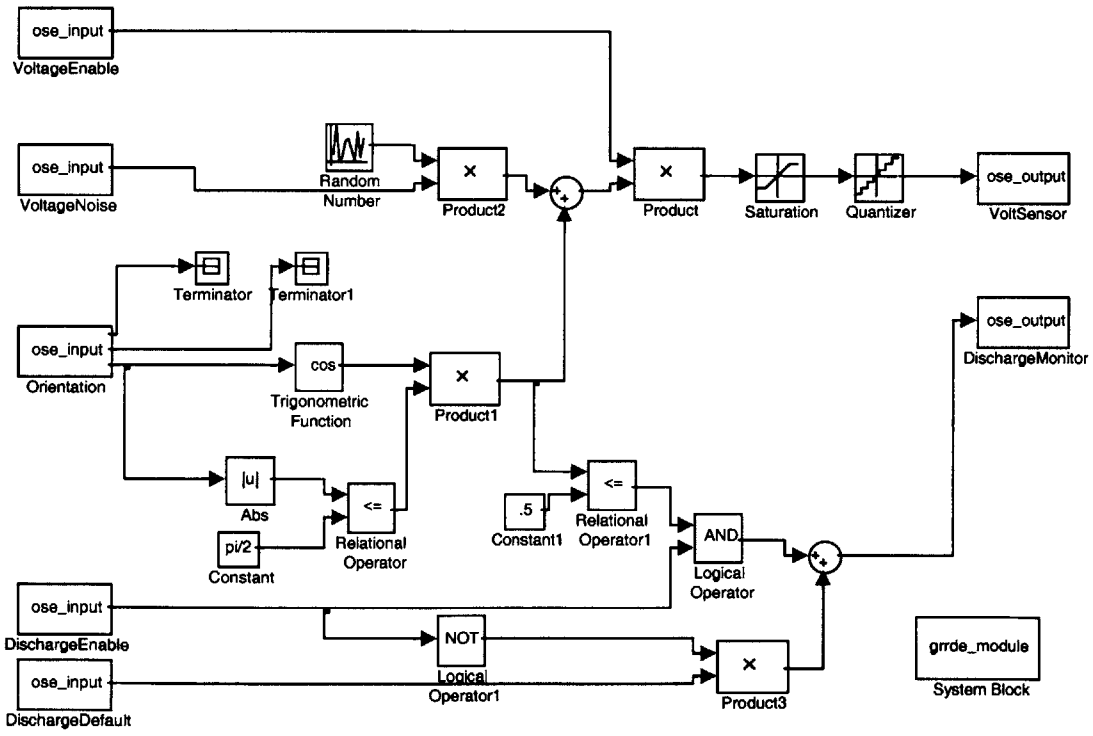


Figure 9.3 Block-diagram of the electrical simulator.

ules using the tools described in Chapter 8. The dynamic simulator consists of a basic propagator plus an extra set of components to add noise and quantization effects to the

sensor readings. The electrical simulator reads truth values from the dynamics simulator and calculates noisy values for the two extra sensors. Additional inputs to the electrical simulator allow testers to selectively adjust the sensor noise or to force the sensor readings to one of its extremes.

The Marple model for our simple spacecraft is shown in Figure 9.4. It consists of five components and six sensors. We note that the SCDynamics component is not a physical device. Marple models frequently employ *pseudo-components* such as this to model system dynamics. In this case it relates the effect of wheel-torque to spacecraft rotation. We can consider the constraints in each component separately.

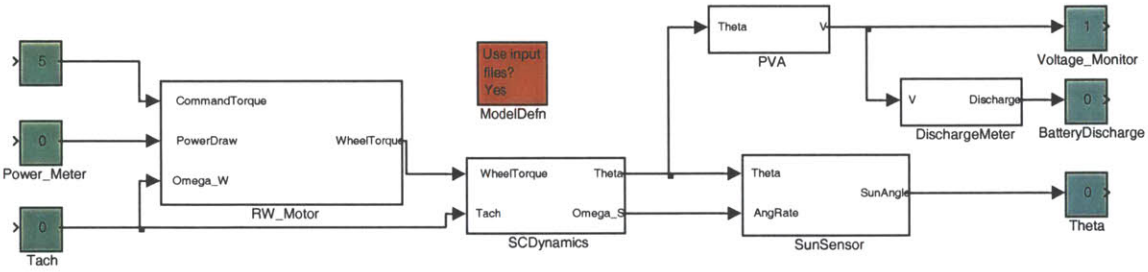


Figure 9.4 Marple Model of the Simple-Spacecraft.

RW_Motor

This component represents the reaction wheel and motor assembly. We monitor the commanded torque τ_c, the power draw P_w, and the wheel speed from the tachometer ω_w. The output node contains the torque applied to the wheel τ_w. The following constraint equations are used:

Under nominal conditions, the output torque should equal the input torque:

$$\tau_c = \tau_w \tag{9.1}$$

We can relate the average power draw to the average wheel speed between this timestep and the last[1], and the applied torque (this is an ideal, mechanical power):

$$P_w = \tau_w \cdot \frac{(\omega_w + \omega_{w_0})}{2} \tag{9.2}$$

If $\tau_w \neq 0$, then we can provide two constraint equations for ω_w, one from the above equation, and another:

$$\omega_w = \omega_{w_0} + \frac{\tau_w}{I_w} \tag{9.3}$$

We recall that I_w represents the moment of inertia of the reaction wheel.

Eqns. 9.2 and 9.3 can be rearranged to provide forward constraints on the value of τ_w. We omit them for the sake of brevity.

SCDynamics

This pseudo-component model represents the dynamics of the spacecraft system. It relates the wheel speed ω_w, the satellite speed ω_{sc}, the sun angle θ, and the wheel torque τ_w. We do not allow this component to fail.

The constraints are:

$$\theta = \theta_0 - \omega_s \cdot \Delta t + \frac{1}{2}\frac{\tau_w}{I_{sc}} \cdot \Delta t^2 \tag{9.4}$$

The satellite speed can be estimated from the torque:

$$\omega_{sc} = \omega_{sc_0} - \frac{\tau_w}{I_{sc}} \cdot \Delta t \tag{9.5}$$

and from the change in θ:

$$\omega_{sc} = \frac{2(\theta_0 - \theta)}{\Delta t} + \omega_{sc_0} \tag{9.6}$$

1. Marple models can keep a history of previous sensor values. In our example, quantities from the previous time-step are subscripted with a zero.

The wheel reverse constraint for the wheel torque is found by inverting Eqn. 9.5. Also, assuming zero bias momentum:

$$\omega_w = -\frac{I_{sc}}{I_w}\omega_{sc} \tag{9.7}$$

PVA

We assume a very simple model for the solar array. We estimate the array voltage as:

$$V = \cos(\theta) \tag{9.8}$$

We do not provide a reverse constraint in this model, since this would introduce a sign ambiguity. The current version of the Marple shell cannot handle values with ambiguous sign.

Discharge_Meter

This component has a very simple model. The boolean (i.e. *true* or *false*) discharge indicator variable D is given by:

$$D \Leftrightarrow V > V_T \tag{9.9}$$

for a fixed threshold value V_T.

Sun-Sensor

This component measures the angle between the satellite and the sun. It is modelled as a component and not just a sensor, so that we can estimate the rotation of the spacecraft from difference in angles. The actual *theta* sensor reading is modelled as infallible. The spacecraft rotation rate is estimated from:

$$\omega_{sc} = \frac{2 \cdot (\theta_0 - \theta)}{\Delta t} - \omega_{sc_0} \tag{9.10}$$

9.3.2 Software Architecture

During development the Marple engine was converted into a form suitable for execution on GFLOPS's embedded processors. Our prototype architecture for this model is shown in Figure 9.5. Incoming sensor readings were provided by periodic contracts operating at 1Hz. The system model is compiled into a C++ object.

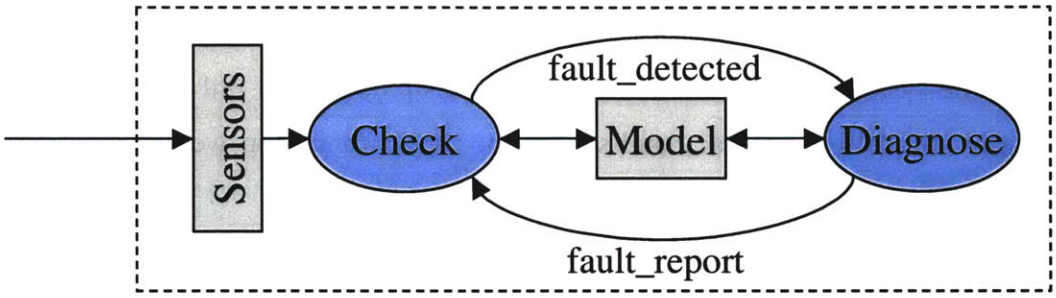


Figure 9.5 The Marple module architecture. GRRDE contracts provide sensor readings for model checking processes.

Two processes share a copy of the system model. The first process is the model checker. Once per cycle, it will update the model with the incoming sensor values and check for consistency. If conflicts are discovered, it will suspend its own update functions and trigger the diagnostic process. The *diagnose* process only executes when it receives this fault notification. It attempts to isolate the fault and will then report back its findings to the *checker* process. When the fault report is returned, the checker restarts its monitoring.

This separated execution is not required for our simple model. In a more complex system, however, the checking consistency is substantially faster and more deterministic than diagnosis. Therefore, the consistency check can execute at a medium priority. Since the fault isolation process takes longer and is not deterministic [Kolcio, et al, 1999], it is placed in its own low-priority process and runs in an aperiodic fashion. Furthermore, a larger system may require that the checker continue to perform various functions, such as sensor filtering, while the diagnosis is proceeding. Determining the appropriate system response after a fault isolation report is a question of overall mission operations policy.

Several options exist ranging from putting the spacecraft into a 'safe' mode, to suspending the contribution of the failed component and continuing nominal diagnostic activities. For our purposes, it is sufficient that we log the report and reset the model[1].

9.3.3 Observations

A number of tests were conducted with the simulation. Test descriptions and diagnostic results are summarized in Table 9.1. Although the static diagnoses in the event of failure were fairly accurate, false alarms were observed when highly dynamic behaviours were present in the system. Careful revision of the models reduced, but did not eliminate, the false-alarms. Modelling accuracy and subscription synchronization were key influences on performance.

TABLE 9.1 Autonomy Test Results

Control Activity	Injected Fault	Diagnostic Result
Idle	None	No faults reported
Active (hold position)	None	No faults reported
Active (sun pointing)	Voltage sensor fails ($V = 0, D = false$)	Correctly identifies failed sensor
Active (sun pointing)	PVA fails($V = 0, D = true$)	Correctly identifies failed component
Active (slew)	None	Transient false alarm occasionally reported during high acceleration
Active (sun-pointing)	Noisy Tachometer	Marple consistently suspected the tachometer, but would also often suspect RW and Sun-Sensor
Active (sun-pointing)	Noisy Torque actuation	Marple model very insensitive to torque noise. Caused by large tolerances due to sensitivity to Sun-Sensor sensitivity.
Active (slew)	Sun-sensor fails (very noisy)	Occasional false alarms at start of slew, but accurate diagnoses after fault in sun-sensor.

1. In a more sophisticated system we would likely reset suspended models periodically to see whether failures were transient. If the same diagnosis repeats itself, the component might be suspended permanently.

Our Marple model was revised several times during development. We had to relax several of the tolerance functions so that nominal sensor noise did not inadvertently cause a false alarm. This behaviour was attributed to the amplifying effect of some of the constraint equations (e.g. small θ errors translate into large predicted torque errors). When the tolerance functions were relaxed, the model would still identify gross errors in the sensor readings (e.g. saturated, zeroed) but small biases and noise were ambiguous.

Noise also played a role during dynamic operations. During testing, we experimented with time-averaging the sensor inputs to reduce the effects of noise. This was done before updating the model. Results were varied: if too many samples were included, Marple models would not track dynamic effects; too few, and noise would trigger failure-alarms. This points to a coupling between the control system design and diagnostic efficiency.

Close temporal alignment (see Section 7.2.3) between software components was essential to accomplish correct Marple operation during dynamic activity. If the torque commands and sensor data are not reconciled with the values used by the controller, the model predictions will be erroneous. These effects can be minimized by careful synchronization and good software design. A better simulation design would use sensor filter modules in the flight software. Both Marple, and the controller, would use the same noise-filtered data. Since this would reduce the need for synchronization between flight software and the simulator, fewer artificial constraints are placed on the flight software. Without real-time service guarantees, ensuring data consistency would be much more difficult.

One of the primary goals of this study was to demonstrate the layering of functional capability. Capitalizing on the information mobility offered by GRRDE, we were able to integrate a fault diagnosis module into a simulation without disturbing the low-level flight software. Ideally, we would like to perform this type of integration with any existing system. The problems that we encountered suggest that this integration may require significant effort for arbitrary, underlying designs. However, if developers foresee this functional layering during the design of the low-level software, they can, with minimal effort, make

provision for its later inclusion. Thus, GRRDE is best used to preserve the option of using a software technology whose maturity is in question at the beginning of a mission.

9.4 Future Work

Integrating Marple with GRRDE was generally effective. The autonomous capabilities offered by this system supplemented the existing flight software, and the middleware allowed access to the system's state information with minimal disturbance. We suspect that the observed testing effectiveness was more indicative of the simplicity of the modelled system, rather than a clear measure of real-world performance. Although this study shows the promise of GRRDE to support autonomy, there is room for additional study. We divide these recommendations into general expansion of the Marple diagnostic engine, and a discussion of using GRRDE for other autonomy related applications.

9.4.1 Expanding the MARPLE Study

The feasibility of integrating autonomy applications with GRRDE has been established, but a more detailed study would provide better insight into using Marple in a diagnostic setting. We recommend an expanded study to assess the issues involved in performing diagnosis with a non-trivial system model. Selecting a complex system to model would allow more concrete conclusions. We foresee the following benefits. First, greater familiarity with the modelling technique would produce higher-quality models. Second, the Marple executable would benefit from thorough reengineering for embedded applications; time did not permit more than a cursory conversion during this study. Third, a more elaborate system model suggests more elaborate flight software. Thus, an expanded study could be used to develop guidelines for efficient integration with GRRDE.

9.4.2 Other Autonomy Opportunities

Opportunities exist to explore the integration of other autonomous capabilities into GRRDE-based simulations. This research might progress in one of several directions. The

simplest concept is to explore how other types of autonomy technology can operate in the GRRDE environment. A less obvious direction is to incorporate autonomy techniques directly *into* GRRDE, enhancing the services of the middleware.

We have limited the scope of our study to one particular model-based technique. Other diagnostic systems have been developed for spacecraft systems. The Livingstone [Williams & Nayak, 1996] package is a qualitative diagnostic and repair tool that was flown on the Deep Space-1 spacecraft. Since its previous flight experience, Livingstone has been updated and revised. Titan [Williams, et al, 2002] combines diagnosis with other autonomous technologies such as execution. An interesting feature of this system is its close integration with other autonomous tools.

Health monitoring is a primarily passive task. Many active roles for autonomy have also been envisioned. Observation and maneuver planning, as well as sequencing and execution have all been investigated by various studies. Implementation of these tools in a series of GFLOPS simulations would expand our knowledge of how autonomy and middleware can interact. Substantial effort would be involved, since evaluating these interactions with GRRDE would require the parallel development of more complex simulations.

Instead of simply using autonomy with GRRDE, in direct support of a mission, we might also consider adding autonomy to GRRDE. We might envision a low-overhead diagnostic utility which we could add to each software module so that each component and subsystem intelligently reports its own health. Alternatively, a specialized distributed diagnostic utility could be used to automatically monitor and maintain subscription connections in the system. In the event of a temporary communications interruption, this tool would attempt to restore software connections. These possibilities are intriguing, but we must take care to select services that are widely useful and efficient to implement.

9.5 Summary

This chapter has explored the use of GRRDE to support the integration of autonomous fault diagnostics with pre-existing flight software. Although the results from our initial study are promising, a more elaborate example is needed to gauge the full impact of this technique. These results suggest that while GRRDE can be used to layer functionality atop of arbitrary low-level software, maximum benefits are attained when provisions are made in the underlying software to make useful data readily available.

This is the second of our two technology limited technology demonstrations. In Chapter 10, we explore the use of GRRDE in the development of larger more complex applications.

Chapter 10

THE TECHSAT 21 GFLOPS SIMULATION

The examples presented so far have been simple in design. Each study was chosen to illustrate a particular application of GRRDE and show how middleware can interact with other software engineering technology. In this chapter we examine the application of GRRDE to a larger, more complex simulation. We must manage interconnections between many software modules, developed by different people, serving very different functions.

The mission chosen for this study is the TechSat 21 technology demonstrator mission. TechSat 21 is a distributed aperture radar concept. Our simulation scenario begins with the precision formation flight of a small cluster of satellites. As the target area of Earth comes into view, the satellites must activate their radar transmitters, coordinate reception, exchange the returned signals and synthesize the ground scene. Flight software for this simulation must be capable of this wide array of tasks.

10.1 Overview

The simulation undertaken in this study was our largest application of GRRDE. The TechSat 21 mission concept is quite complex, and many operational questions are still unanswered. We hoped that the use of GRRDE will help develop a robust simulation, capable of examining some of these operational issues. This study was also a chance for self-reflection. Smaller simulations, although insightful, do not possess the complexity

necessary to truly require middleware communications. It is our intention to use this large simulation to evaluate our proposed design methods and identify any areas of deficiency.

The main contribution to system complexity is simply the size of the development effort. Simulating TechSat 21 required many simulator modules, many flight software modules, and many interconnections. Each of these components requires a substantial amount of software engineering. To ease the burden of creating these modules, this simulation was developed concurrently by several people. Each of the three team members was responsible for several software components. A larger engineering team accelerates the software development, but makes interface management more important.

We begin with a summary of the TechSat 21 mission. This helps establish the background for our study. This is followed by a discussion of the preliminary simulation design. We examine both module design and interface specification. Where appropriate we also consider implementation issues and component-level testing. We conclude with an examination of the complete simulation, and our observations concerning software engineering with GRRDE. Several important issues are raised that highlight directions for future development of the GRRDE middleware.

10.2 The TechSat 21 Flight Experiment

Perhaps one of the more ambitious distributed satellite systems proposed is the radar system known as the Technology Satellite for the 21st Century (TechSat 21). TechSat 21 seeks to validate the feasibility of a number of technologies aimed at making space systems smaller, cheaper and more reliable. Even the chosen mission is a demonstration of technology. Using a cluster of four to twenty satellites in Low Earth Orbit (800 km.), TechSat21 will use advanced techniques for *ground moving target indication (GMTI).* While traditional approaches to space-based radar have required huge antennas, TechSat21 seeks to exploit the science of sparse aperture arrays, using antennas of only a couple of metres across. The coherent processing and the use of transmitter and receiver diversity allows signal gains of 100-1000 or even more [Das & Cobb, 1998]. While this

figure is encouraging, it levies stringent requirements on atmospheric propagation modelling and on-board processing.

The demanding processing requirements arise from the sheer volume of data that are collected. The aggregate rate of data collection is on the order of 10^{10} bits per second [Enright, et al, 1999]. If we wish to avoid on-board processing, this information must be conveyed to the ground. This suggests two possible implementations, both of which have undesirable features (Figure 10.1).

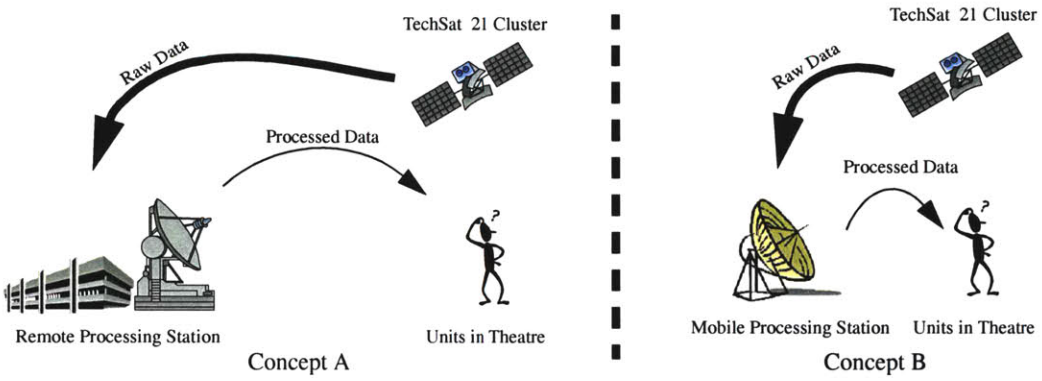


Figure 10.1 Alternatives to on-board processing. Remote processing station (*A*) or Local downlink (*B*).

A remote processing station offers security and processing power, but requires a data relay network both from the satellites, and back to the battlefield. On the other hand, local downlink to mobile stations requires deploying advanced and bulky hardware in potentially hostile situations. Neither of these approaches seems particularly attractive. The difficulty stems from supporting high bandwidth communications channels in adverse conditions. Performing radar processing within the cluster of satellites has the potential to drastically reduce the required downlink requirements. For this strategy to be effective, the satellites require unprecedented processing capabilities and coordination.

10.3 Design and Implementation

Several variations of the TechSat 21 concept are being reviewed. The preliminary mission involves a separated aperture radar experiment involving one cluster of three satellites.

Processing in the experimental mission will be performed on the ground. This mission will validate the sparse-aperture space-based radar concept for application to a full constellation of satellites. A follow-on mission would involve approximately forty clusters of 4-12 satellites and provide simultaneous, global coverage. The ideal processing architecture for the full system has not yet been determined. Our simulation implements a hybrid concept. We consider a cluster of four satellites and assume that the processing is performed on-board. Unlike the full system, we consider only a single cluster in isolation. Coordination and communication with other clusters are ignored.

This section details the engineering of the TechSat 21 simulation. We begin with a discussion of the system scope and the primary communication pathways. From this we examine the interface specification process. The last part of this section describes each of the functional modules individually. We have tried to tailor the simulation to the general sense of the hardware design, however, some inconsistencies may be found.

10.3.1 Simulation Scope

At the start of simulation development, our first task was to determine the scope of the study. Without the full personnel of a flight software development team, prudence demanded that we select only the most relevant subsystems to model. The TechSat21 mission is centered around its radar experiment; system functions only loosely coupled to the payload were culled from the simulation plan. The list of candidate components is shown in Table 10.1.

After careful consideration we arrived at the software design shown in Figure 10.2. All essential subsystems were included. Additionally, attitude control was also integrated since preliminary development had already begun. This diagram shows the distinctions between simulator code and flight software. To facilitate the prototyping of the system, the simulator modules were installed on the idle embedded processors rather than on the PCs. The user interface modules remained on the PCs. Our initial design called for a specialized user-interface for constellation operators. This effort was de-scoped from the final design

TABLE 10.1 Relative Importance of TechSat21 Subsystems

Subsystem	Rank	Remarks
Propulsion/Orbit Control	1	Necessary for formation flight
Radar	1	Central to mission
Attitude Determination	1	Required for radar processing
Inter-satellite Communication	1	Functionality needed (not necessarily hardware representations)
Down-link	1	Functionality needed (i.e. get processed data out of satellites)
Attitude Control	2	TechSat is primarily gravity gradient stabilized. Tight control unnecessary
Power	2	Necessary, but behaviour is uninteresting
Structural	3	Passive system
Thermal	3	Secondary system
Ground Station	3	Distracts from focus on spaceborne processing
Computational Architecture	3	Can't effectively represent specialized hardware architectures.

due to time constraints and because it duplicated many functions of the other clients. Some of the interfaces to the command sequencer and fault management modules were defined but these functions have not been implemented.

The baseline simulation design relied on three central capabilities: radar, orbit control, and attitude control. Each of these three divisions contains elements of simulation, flight software and user interface. Detailed design began once the simulation scope was established.

10.3.2 Interface Specification

Simulation design with GRRDE relies on identifying published state information in each module and identifying the data-flow architecture. Initial design iterations considered these communications fairly generally. Of primary concern was simply identifying where

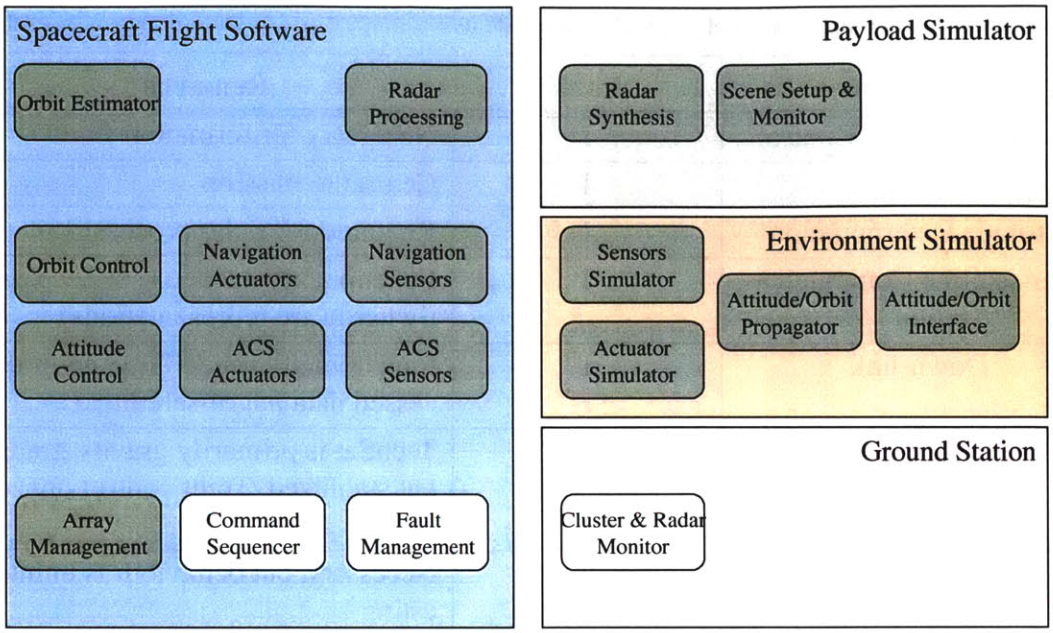


Figure 10.2 Software Functional Decomposition for TechSat 21 simulation. Dark boxes show baseline modules, while light boxes represent possible future improvements.

the connections were. Once the functional allocation to the modules was clarified, we were able to provide more detailed specifications of the module interfaces.

We can identify data flow for each of the three chief software tasks. Sketches of this data flow are shown in Figure 10.3. We observe that most of the connectivity remains within each spacecraft. This structure is an intentional part of our modular design. Limiting the number of interconnections between spacecraft conserves bandwidth and makes the interactions easier to analyze. We make the following observations about the general patterns of data flow:

- Attitude control is almost entirely a local activity. Some coordination with other spacecraft or the ground might be present, but current hardware designs rely mainly on passive gravity gradient stabilization. The spacecraft's microthrusters can be used for attitude control, but the attitude control is not a high-performance system.
- Orbit-control's reliance on satellite collaboration is a systems-level issue. The spacecraft can attempt to maintain absolute orbital elements supplied by the ground, or alternately, they may work together to manage their relative

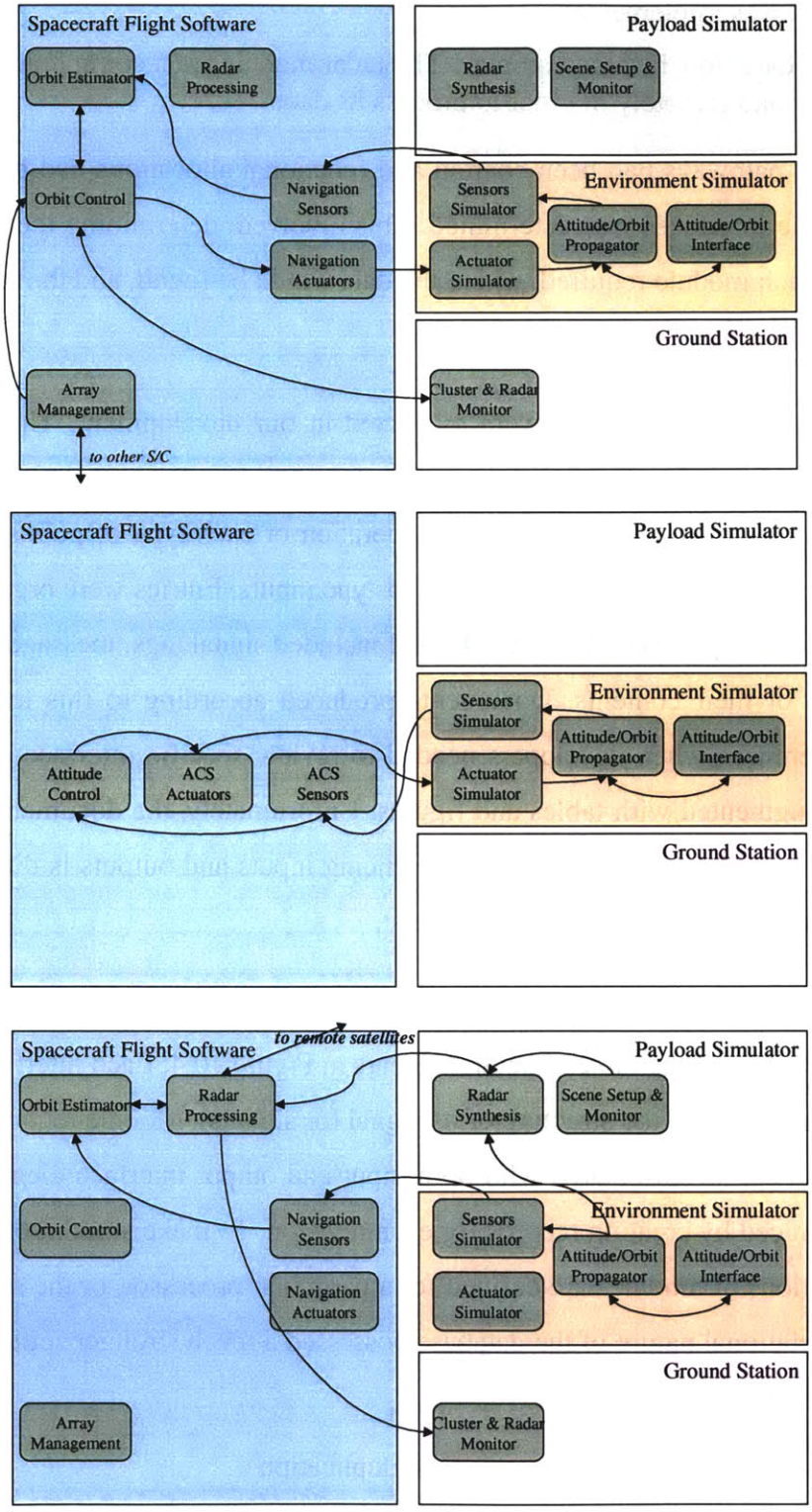


Figure 10.3 TechSat 21 Information flow. Major data flow for orbit-control (top), attitude control (middle), radar processing (bottom).

spacing. Some interfaces for collaboration are in place, but only the absolute control was implemented.

- Radar data flow is quite complex. The radar module must communicate both locally and remotely in order to process its data.

Once the data pathways had been charted and functional allocations had been made, the structure of the interfaces was determined. This involved: determining the types of state information each module required, where the data would be found, and the temporal characteristics of the exchanged data.

Two methods of documentation were evaluated in our development. The first method evaluated was a structured, manual template. An example of this technique is given in Appendix B. This method involved the enumeration of each type of published data produced by the module as well as any command-type inputs. Entries were organized by service name and data product (Section 4.4), and included signal tags, message structure and a description of their contents. Documents produced according to this technique were excellent references when developers needed to review specific interface elements, and were easily augmented with tables and figures. Unfortunately, the documents themselves provide little semantic structure. Cross referencing inputs and outputs is difficult and the big-picture of the whole simulation can be lost.

To address these issues we developed a database application to track module interface definitions. A diagram of the table structure appears in Figure 10.4. Each interface element is defined by its usage and the structure of its signal (or signals). Module records are defined separately and can be associated with both input and output interface elements. Module evolution is traced by creating records for each revision. To make simulation composition easier, collections of modules are defined for a particular processor, or the simulation as a whole. The relational nature of the database possessed a much stronger semantic structure and allowed us to:

- Manage signal tags to protect against duplication
- Check for consistency in a simulation configuration (i.e. are all inputs and outputs accounted for?)

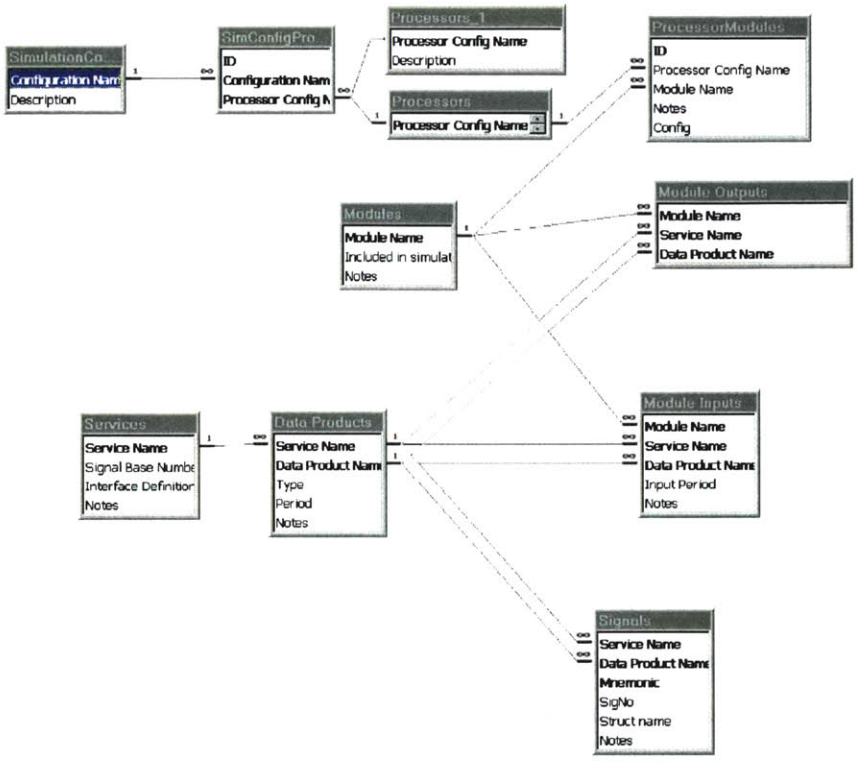


Figure 10.4 Interface Database Structure.

- Generate module documentation. Useful module-level interface references are automatically compiled from the signal-level documentation
- Generate C header files. This facility was only partly explored, but was useful when creating lists of signal tag definitions for header files.

One minor drawback in applying this approach was the difficult of adding free-form documentation like reference tables or diagrams. We suspect that this was primarily due to our inexperience with database design, rather than a limitation of the technique. Overall we found the database aided our efforts, both during initial design and later, during implementation, when slight changes were made to the interfaces.

10.3.3 Simulation Modules

This section briefly describes each of the modules in the simulation. Where appropriate, we remark on the testing of the module.

Navigation Sensors

This module represents an interface to hardware. It accesses the navigation hardware and provides spacecraft position and velocity in the Earth-Centered Inertial (ECI) reference frame. Presently, there is very little intelligence in this module, since our initial testing assumed a benign noise environment and a simple simulator interface. Sensor data is available from the simulator directly in ECI values. This is not completely unreasonable since GPS can supply this type of data. In a more mature system the module might include a Kalman filter or a similar algorithm to remove data noise or perform sensor fusion for a less direct set of sensors.

Sensor Simulator

This is the other side of the interface for the navigation sensors. It receives true position data for the satellites and adds appropriate sensor noise or quantization to the values. In the current incarnation of the simulation both input and outputs are in ECI coordinates. The magnitude of the noise can be set by an external command and is customizable by satellite and quantity (e.g. position or velocity). By default simulations assume zero noise.

The sensor simulator also processes the truth data destined for the attitude sensors. The propagated quaternion attitude and angular velocity can be transformed as desired into appropriate sensor readings. The current simulation configuration does not transform the attitude data, it simply adds a selectable amount of noise to the readings.

Orbit Estimator

This module provides navigation interpolation and extrapolation. The spacecraft's navigation sensors operate at 10Hz., but in the interval between readings, they can move about 750m. Correct operation of the radar processing algorithms require position knowledge of much greater accuracy. The system monitors incoming navigation data and maintains a linearized model of orbital dynamics, including J2 effect [Schweighart & Sedwick, 2002]. Upon request, the module will calculate the spacecraft's exact position at a specified time.

Orbit Control

Satellites in orbit are perturbed by various external forces such as drag, asymmetric gravity field (e.g. J2 and J4 effects), and the gravity of the sun and moon. Since the imaging effectiveness of the satellite array depends on maintenance of the cluster formation, propellant must be expended to counteract some of these effects. The orbit control module gathers navigation data from the sensors and calculates the appropriate thruster firings. Presently, the orbit control algorithm is purely localized. The ground provides a reference orbit for each satellite and the controller tries to maintain that orbit. A potential improvement would consider a more fuel efficient approach that emphasizes maintaining relative cluster orientation, but allows drift in the cluster as a whole. A sample of the controller performance is shown in Figure 10.5. The figure charts relative displacement, in the rotating orbital frame. The origin is the cluster centre and the blue curve represents the reference path of the satellite.

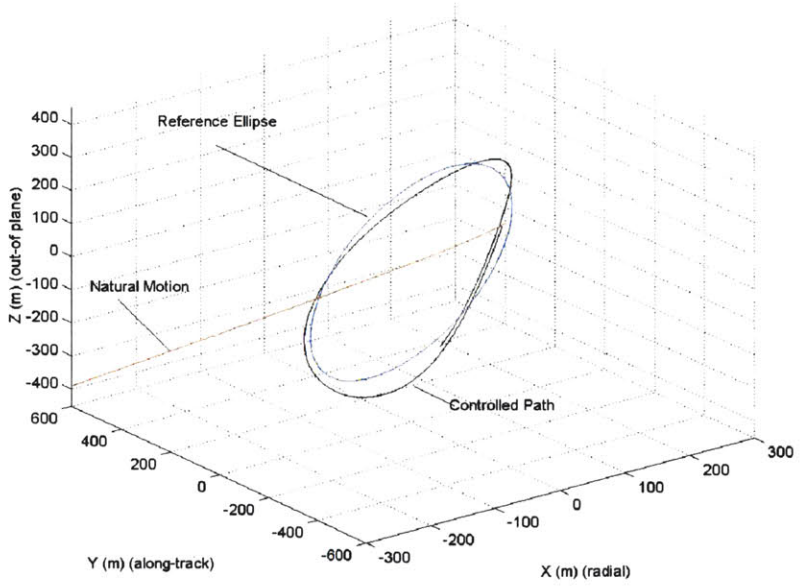


Figure 10.5 Demonstration of Orbit Control from Real-Time Simulation.

Actuator Module

The orbit controller calculates the requested thrust actuation in the orbit frame. Likewise the attitude controller calculates requested torques in the body frame. Since TechSat 21 is designed to use pulsed micro-thrusters, two operations must be performed on the thruster commands. First, the thrust and torque vectors must be reconciled and mapped to the geometry of the body mounted thrusters. Second, the thrusters must be pulse modulated to provide the required net impulse over the control time-step. Our initial design called for two distinct actuator modules, but the functions were combined when we realized that the same thrusters would be used for both purposes. The impulse outputs from this module are sent to the actuator simulator.

Actuator Simulator

This is the counterpart to the previous module. It receives thrust commands and converts them into inertial forces and body-referenced torques. The resulting updates are sent to the propagator module. Non-idealities may also be added in the form of thruster variability, mean impulse, failure, or misalignment. In most of our simulations, we assumed perfect thrusters.

Attitude Sensors

The role of this module is to interface with the sensor simulator. Presently, the sensors deliver quaternion attitude and body-reference angular velocity, but future versions may require sensor fusion and more elaborate state estimation.

Attitude Control

Our current state propagator does not model many attitude disturbances and the TechSat 21 spacecraft have fairly loose pointing requirements. Thus, our attitude control requirements are fairly modest. The implemented method is based on the quaternion feedback described in [Wie & Barba, 1984]. Some large-angle test maneuvers are shown in

Figure 10.6, but most of the time, the ACS is only used to supplement the gravity gradient effects and keep the satellites pointed at the ground.

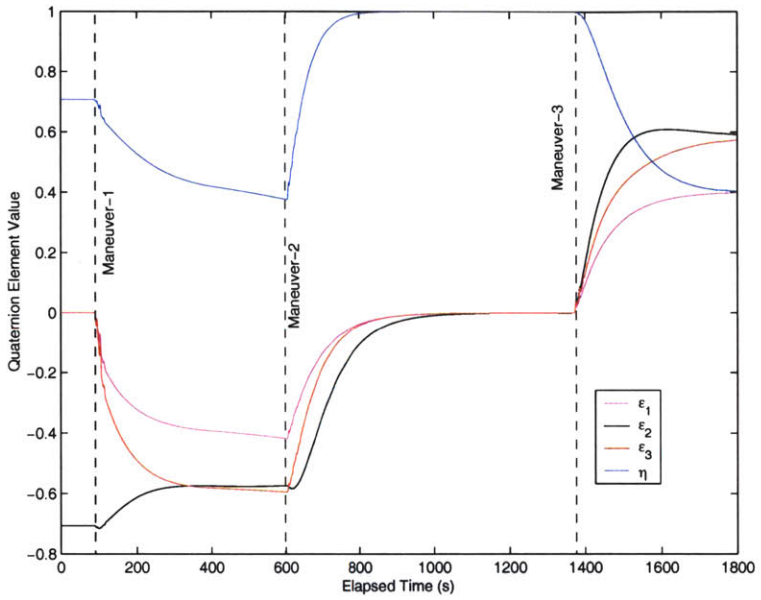


Figure 10.6 Time traces of quaternion attitude during maneuvering.

Orbit and Attitude Propagator

This is the workhorse of the dynamics simulation. It propagates the non-linear state equations and adds the effects of disturbances. Spacecraft maneuvers are accounted for by considering impulsive changes to the state. The propagator combines a sixth-order Runge-Kutta integrator with a high-accuracy interpolation function to provide very accurate state evaluation at only modest computational cost.

Dynamics User Interface

This module allows the user to monitor and configure the simulation dynamics. This application runs on the PCs and has a range of capabilities: ground tracks and satellite attitude can be displayed, results can be logged to files, and simulation parameters can be set manually or from user-defined scripts. Sample screen-shots are shown in Figure 10.7 and Figure 10.8.

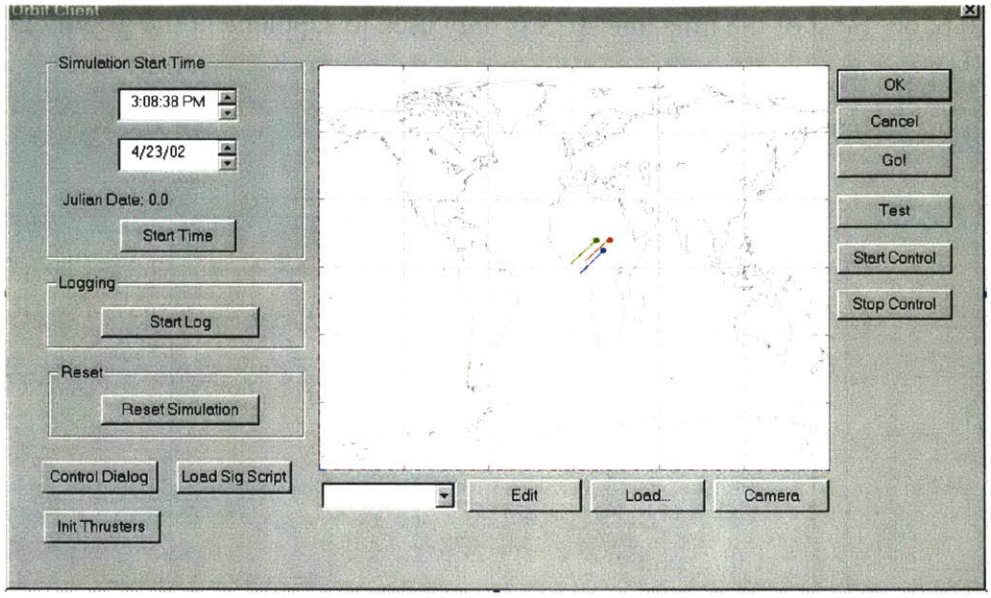


Figure 10.7 Main user-interface to simulation operation.

Figure 10.8 Three-Dimensional Satellite Visualization.

Radar Interface

Like the previous example, this PC application is used to configure and monitor the radar operations of the simulation. Typical scenarios involve a number of targets moving in a specified area. The user defines the ground scene using this interface, which will automatically send the appropriate initialization to the radar simulator. During operation, the simulator monitors the processing results of the satellites and overlays their target predictions

with the true locations. A sample screen-shot is shown in Figure 10.9. Supplemental tools were also included to automatically generate overflight scenarios for the dynamics simulator.

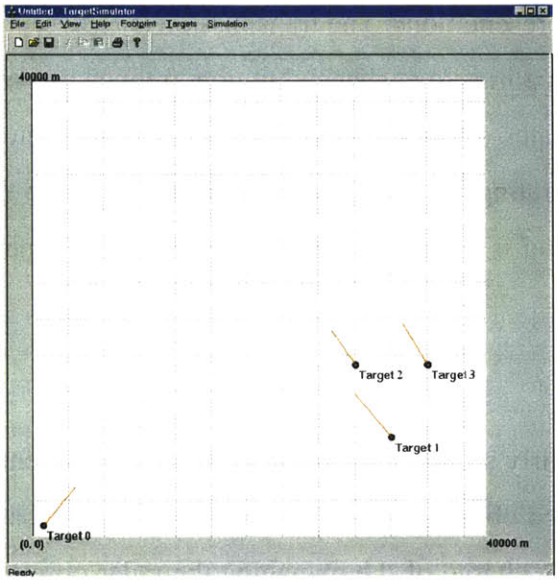


Figure 10.9 TechSat 21 Radar GUI.

Initially, we had planned to include a separate GUI to represent an operator's (rather than omniscient) view of the running simulation. The importance of this application was considered low since, much of the desired functionality (e.g. comparing truth with estimates), was already present in the other applications.

Radar Simulator

This module provided the bulk of the payload-related simulation. The basic operating concept of the TechSat 21 system is to use one satellite to transmit a radar pulse. When the pulse returns, it is received simultaneously by all of the satellites in that cluster. Slight differences between each return signal allow target locations to be pin-pointed.

When prompted by a radar pulse from the satellite, the simulator would synthesize return values. Each spacecraft in the cluster would receive a response corresponding to its own

signal return. These are calculated precisely from the viewing geometry. The signal contains reflected contributions from the targets and the ground (clutter).

A number of idealizations were made about the operation of this module. Since simulating the front-end of the radar system would be difficult, both the emitted pulse and the return signals have abstract representations. The transmission is represented with a structure that describes the essential qualities (e.g. shape, duration, frequency). When the simulator generates the returns, it creates a base-band version of the signal return, rather than the full modulated carrier. Generating the full waveforms in software for either case is a waste of simulation bandwidth and is unrealistic. These steps would be performed in specialized devices.

Array Management

This module served a fairly simple purpose. It performed rudimentary array coordination and ensured synchronized observations. Upon initialization, the satellites would choose a 'leader' that would be in charge of the managing the radar observation. The operator on the ground must designate a desired imaging location. The lead satellite consults the orbit propagator, and calculates when the target area will come into view. As the viewing time approaches the array manager will tell the transmitting spacecraft when to activate its radar, and tell the other spacecraft to prepare for incoming data.

Radar Processing Module

The radar module is responsible for the collection, exchange, processing and reporting of the radar returns. The processing approach chosen for this module is known as the Scanned Pattern Interferometric Radar (SPIR) algorithm [Marais, 2001]. During the collection stage, each satellite accepts data from the radar simulator. In order to synthesize the ground scene, all of the satellite signals must be combined. The radar returns are divided into time segments or *snapshots*. A snapshot represents the period of time over which changes in viewing geometry are small enough that we can coherently combine the signals. Each snapshot required several megabytes of storage on each satellite. We initially

considered breaking each snapshot up into several pieces (i.e. one for each satellite), but instead adopted the simpler approach of processing a whole snapshot on a single satellite. Thus, during the exchange phase, all satellites would send their snapshot data to a designated recipient. Processing duties alternated with each snapshot. Finally, when the signals were combined, the results processed and the targets identified, the processing satellite would make the data available to the ground.

Difficulties were encountered in the implementation of the radar processing. The mechanics of the system operation (reception, exchange, reporting) were all implemented successfully. Unfortunately, the radar processing was ineffective. Reliable target discrimination was not achieved and the few correct identifications observed were too inconsistent to be trusted. We suspect that these difficulties were due to the immaturity of the SPIR algorithm. Previously, the algorithm had been evaluated for one-dimensional observations, but the two-dimensional case had not been validated. The departure of the team's radar expert made further progress impossible and development was soon halted.

Although these setbacks had a negative impact of the functional effectiveness of the TechSat 21 simulation, the GRRDE-related goals had already been achieved.

10.4 Discussion

Difficulties encountered in radar processing prevented a full, functional evaluation of our TechSat 21 simulation, but our observations of the development process and of the state of the final software allow us to draw several conclusions about the use of GRRDE in complex systems. During many phases of development the GRRDE-based approach facilitated design and implementation. Other efforts were neither helped nor hindered by GRRDE. Finally, in a few isolated situations, we found that the framework actually impaired the development process. This section reflects on the overall success of the TechSat 21 study and closely examines the utility of the GRRDE services in different situations.

Structurally, the flight software is complete. Simulations containing all the implemented modules can be composed and scenarios can be loaded. Every component in the TechSat 21 simulation can communicate effectively with its peers, and the structure and timing of each message agrees with design. That the contents of some of these messages (i.e. the identified targets) were incorrect does not diminish this success. Thus, taking a global view, GRRDE is clearly effective in binding software modules together. These observations also extend to many of the details of development.

Modular GRRDE design, together with explicit state representation, greatly assisted the shared development process. The time spent cataloguing the interface definitions and GRRDE communications services made integration of separately developed components relatively straightforward. Most problems that we encountered during integration were due to deviations from the interface standards, rather than any fundamental incompatibility. Elaborate control systems and simulation monitors were significantly aided by periodic contracts. The orbit and attitude control systems made extensive use of these services. Similarly, aperiodic contracts were very effective for command feedback, sequencing and status indication. The flexibility of the architecture allowed software subtasks (e.g. orbit control) to be tested in isolation from the other components. This agrees with our intuition about GRRDE service design.

Our simulation development uncovered several forms of interaction that did not map well onto the current GRRDE services. Asynchronous, parameterized query operations such as the state interpolation performed by the orbit estimator, did not behave like either a periodic or aperiodic subscription. Although we were able to define the interface with our methods, the user was required to explicitly manage each request. Likewise, asynchronous commands did not benefit substantially from the GRRDE services. Configuration messages or control set-points are typically implemented as command inputs and outputs.

The second group of awkward interactions involved the transfer of large amounts of data. This effect was seen during radar reception, and during data exchange. OSE signal sizes

are limited to 64KB. To send bulk data, especially to a remote CPU, the communicating processes must be closely coordinated. Flow control must also be considered to prevent overloading the OSE Link-Handlers. These tasks were assigned to subroutines, but the developer was still required to manage many details of the exchange.

These two examples show that where module interactions matched the publish-subscribe metaphor, the run-time features of GRRDE were very useful; where they did not, many communications tasks still had to be performed by hand. Although our middleware still provided secondary benefits, the clear advantages of the GRRDE services were lost.

On a cautionary note, we observed that simulation development was slowed when GRRDE was used at inappropriate times. GRRDE is not a preliminary prototyping tool. During the development of the radar processing module, substantial time was spent attempting to resolve errors originally thought to be superficial. After some time, we transferred the processing code into an off-line application only to discover that the algorithm itself was flawed. Migration to the GFLOPS testbed should only be done once algorithms have been tested directly in off-line environments. This is not a disadvantage of GRRDE, merely a reminder of the importance of good systems engineering.

Since the GFLOPS environment closely mimics flight conditions, several considerations must be kept in mind. First, this is a real-time simulation testbed. We use the actual timing facilities of the operating system and computer hardware to allow developers to assess process interaction and temporal performance. Consequently, a ninety minute orbit takes ninety minutes to simulate. Fine tuning the orbit controller should really be done in a separate environment. Second, there is some overhead in developing for an embedded setting. It takes time to set up processes, to plan for concurrency, to define message handling, etc. Even starting a simulation takes a few minutes. Thus, it is important to formalize interfaces and address embedded concerns during preliminary design, functional validation must be done in a suitable, efficient testing environment.

10.5 Summary

The experiences gained from applying the GRRDE middleware to a complex space simulation are invaluable. Previous application examples have illustrated potential uses of the system, but the experience of actually using the tool permits critical analysis. Our observations from this study highlight clear directions for improving GRRDE. When communications tasks match the publish-subscribe metaphor, the developers' tasks are less complex. Other types of module interaction did not map well to this model and suggest services that could be added to GRRDE. Lastly, some development difficulties underline the importance of judgement in choosing a simulation environment. GFLOPS aids architecture definition and software integration, but the present form is not ideally suited for preliminary software prototypes.

Chapter 11

CONCLUSIONS

The age of distributed satellite systems is close approaching. Some missions have already been built, others are in the planning stages. NASA alone has thirty-five different [Leitner, 2001] distributed satellite missions in the works. And this figure is from a single agency. Commercial, military, and other bodies such as the European Space Agency are all considering various distributed satellite missions. Complexity, an almost inevitable consequence of distributed missions, must be managed if these missions are to succeed.

Flight software complexity is anticipated to be a major challenge in these ambitious missions. Taking inspiration from terrestrial distributed systems, we have developed the GRRDE flight-software middleware as an approach to make distributed flight software development less troublesome, and inherently more reliable.

In this chapter we review the contributions and services of the GRRDE and GFLOPS systems. These contributions are compared to those of other prominent software tools. This reflection, together with the lessons learned in the previous chapters helps to map out the future of the GRRDE system. We consider possible revisions to the middleware itself, as well as opportunities for using GRRDE outside of space systems. A few final remarks then conclude the discussion.

11.1 Summary of Contributions

In this thesis, we have approached the problem of flight software complexity by developing appropriate real-time middleware. Using a high-fidelity, real-time simulation environment, we present three groups of contributions. The capstone of this research is the collection of GRRDE real-time publish-subscribe services. This software product is supported by two secondary efforts. To attract interest in this flight software engineering approach, we have applied GRRDE to both focused and general applications. To encourage adoption, we provide engineering guidelines that offer insight into topics ranging from architectural approaches to real-time analysis. Taken together, these contributions represent a cogent argument for the use of middleware in spacecraft.

11.1.1 Validated Run-Time Services

Safety-critical, real-time systems such as spacecraft rely on strong guarantees of determinism to ensure correct operation. Operating within this context, and aware of the need for high dependability, the GRRDE middleware services were designed and implemented to reduce an engineer's non-productive workload. GRRDE's abstract publish-subscribe services come in two varieties. Periodic subscriptions deliver data at regular intervals and can operate independently from the publisher. This type of software connection is frequently found in control systems. Aperiodic subscriptions function like a multi-cast group communication and are well suited for reporting module status or the like.

Embedded applications demand more from software than just working demonstrations. The GRRDE services were validated with both off-line formal methods and extensive temporal testing and characterization. The algorithms implemented in GRRDE were analyzed using General Timed Automata models. This analysis provided an unambiguous specification of the services provided and demonstrated how the algorithms achieved those goals. Abstract correctness proofs were supplemented with more concrete run-time characterization. The temporal behaviour measured in these tests agreed with the automata

models. Combining these methods illustrates how empirical and abstract techniques can be used to create confidence in software integrity.

11.1.2 Design and Architecture Guidelines

Effective software development with GRRDE depends on the user's understanding of how communications metaphors manifest themselves in the overall process of software engineering. From program start to finish, we provide suggestions designed to maximize the effectiveness of the GRRDE services. Early attention to architectural design and data flow allows the user to identify where GRRDE will be most useful. Explicit state-centric design helps clarify where information is coming from, and where it is going. When moving from prototyping platforms to embedded processors, we provide prosaic instructions on how to physically integrate with GRRDE and predict scheduability. As the design matures further, we suggest ways of handling configuration changes and migration to hardware. Throughout this process, we articulate how common engineering challenges can be viewed from a GRRDE-centric perspective. Having the right mind-set cannot help but improve productivity.

11.1.3 Applications

Our application studies provide concrete illustrations of GRRDE's potential to reduce software interface complexity. Although presented last, our examples are apt to be the first step in convincing a program manager to consider using GRRDE in a mission setting. From the short studies, we see specific benefits; from the large simulation, we effectively demonstrate the benefits of communications abstraction. This process helps to answer the question: "What can this technology do for my mission?"

GRRDE carves a niche for itself, not from adding revolutionary new capabilities to flight software, but by creating a software environment suitable for nurturing emerging technologies. Two popular techniques, automatic code generation and fault diagnosis, integrate well into the publish subscribe framework and benefit from the transparent inter-connec-

tivity. Our larger study shows the value of communications abstraction in managing the integration of complex software.

11.2 Comparative Reflections

In Chapter 2, we introduced several related research and development programs. These programs demonstrate ongoing approaches to distributed embedded software and spacecraft. Having explored the composition of GRRDE in detail, we can now reflect on the relation between these competing programs and our work. Considering the role that GRRDE serves in software development, some tools are potentially complementary, some serve substantially different needs, and others are quite similar. Examining each of these systems helps to define future opportunities for GRRDE middleware.

11.2.1 Object Agent and SuperMOCA

GRRDE has a similar relationship to both Object Agent [Surka, et al, 2001], and SuperMOCA [Jones, et al, 1998]. These systems provide middleware functions including abstract communications services. Although Object Agent and SuperMOCA acknowledge the importance of real-time flight software components, both assume that all real-time activities can be encapsulated within the middleware modules. Consequently, their services are not designed to carry real-time traffic. In contrast, GRRDE explicitly addresses the problem of hard real-time, distributed communications.

11.2.2 Autonomy Test-Bed Environment

The focus of the ATBE [Biesiadecki, et al, 1997] is to facilitate real-time simulator creation. Extensive facilities exist to build highly accurate models and allow them to interact. ATBE has been used to support many missions for both software-only and hardware-in-the-loop testing [Leang, et al, 1997]. The system also supports dynamic reconfiguration of the simulator between simulated components and real hardware. A lot of effort has been expended to support mission testing and validation.

GRRDE, in contrast, is more concerned with improving flight software design. Our primary contributions are the abstract communications services that connect flight software components. That these services allow us to connect to simulators is almost a secondary benefit. We contend that flight software developed with the GRRDE framework would complement the testing capabilities of ATBE. Two issues would have to be resolved before integrating these systems. First, we must decide how the simulator will appear to the flight software (e.g. one big module or many smaller ones). Second, some timing related issues must be clarified. GRRDE employs minimally-invasive timing services using native operating system calls. ATBE supports artificial but flexible check-pointing and 'time-warping' features. Although described in the literature as being "hard real-time", it is unclear how 'hard', the ATBE performance really is. Neither of these issues appear to be show-stoppers, and such integration would be beneficial to all. Flight software development would still exploit communications abstractions, and ATBE's excellent simulation tools would help debugging and testing.

11.2.3 Mission Data Systems

The Jet Propulsion Laboratory's MDS program is attempting to completely revise the process of writing spacecraft flight software. From their architectural principles (or *themes*) [Dvorak, et al, 2000] they are devising a comprehensive approach to flight software development, in an attempt to enable more autonomous spacecraft. GRRDE's emphasis on explicit state specification is directly inspired by the early MDS concepts. Individually, each of their themes would likely be beneficial.

We contend, however, that MDS attempts to change too much, too fast. The change in flight software paradigms seems both technologically premature and politically ill-advised. Software engineering is not yet so advanced that creating complex reliable software is effortless [Leveson, 1992]. Autonomy itself is an even younger discipline. Moreover, when presented as an "all-or-nothing" proposition, MDS risks alienating those program managers who might be willing to accept more moderate innovation at less risk.

Whereas MDS identified their architectural principles and endeavoured to provide a design framework from the top, down, GRRDE acknowledges architectural goals, but proceeds to reach them from the bottom, up. We begin with the state of conventional embedded and flight software, and slowly work upwards.

11.2.4 Real-Time CORBA

CORBA continues to be one of the *de facto* standards for distributed computing applications [Bates, 1998]. It is flexible, it operates seamlessly between many hardware platforms, and its object oriented approach matches popular methods of software development. Real-Time CORBA attempts to define a flavour of the system suitable for real-time applications [OMG, 2000]. Researchers [Schmidt, et al, 1997] have made serious efforts to address real-time performance issues and interest in aerospace applications is growing [Harrison, et al, 1997].

Following the standard practices of the conventional software industry does not necessarily make for good embedded systems. General observations about embedded systems engineering [Wright & Williams, 1993] and specific experience with spacecraft [Stolper, 1999] have suggested that object-oriented approaches may not be ideal.

Although Real-Time CORBA together with the CORBA event service replicates publish-subscribe communications, the system still carries significant overhead which may ultimately be unsuitable for demanding applications. In contrast, GRRDE offers a less comprehensive but less cumbersome tool for highly demanding tasks. For high performance, distributed, hard real-time projects, avoiding unwieldy specification may provide the right combination of embedded performance and service flexibility.

In the long-term opportunities exist to supplement GRRDE's capabilities with extra communications services provided by CORBA. We discuss these prospects in Section 11.3.

11.2.5 Simplex

Simplex and its underlying publish-subscribe middleware is the closest system to GRRDE in intended application and capabilities. Both seek to support embedded software development and both provide deterministic time guarantees. Three factors set GRRDE apart. First, GRRDE supports periodic subscriptions in addition to the more conventional aperiodic broadcast. Thus, GRRDE has provides greater service flexibility. Second, the ability to parameterize dispatch functions allows greater flexibility in message delivery. Third, our algorithms have been formally verified, using automata techniques, to ensure correctness and guarantee temporal properties.

11.3 Future GRRDE Development

The inherent flexibility of software systems, especially software systems that support other software, ensures an unending supply of potential development directions. We have identified three (Figure 11.1) directions for future research. We define *refinements* to be the additional development necessary to turn GRRDE from its present prototype form into a viable system for serious applications. *Enhancements* refer to research into extra capabilities and services not presently included in the GRRDE specification, but judged useful in its role as a middleware systems. Lastly, we consider further *application* studies that would help to garner interest and confidence in the technique.

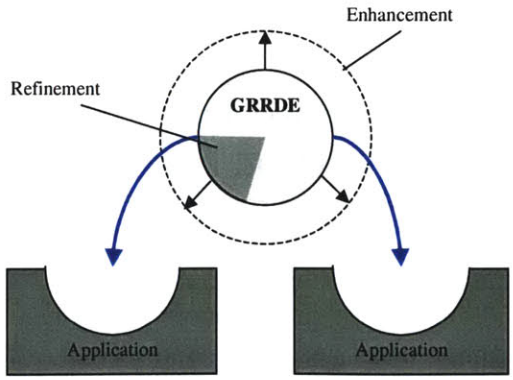


Figure 11.1 Three directions of future work. Refinements define requirements for a commercially viable release, enhancements expand GRRDE services, and applications demonstrate further uses.

11.3.1 Refinements

Although all key technologies are currently implemented in the GRRDE middleware, compromises are always made in research systems. Several matters must be addressed before a general release of the system would be viable. These issues are questions of polishing rather than innovation. Our chief concerns are:

- Optimize change-triggered dispatch mechanism. Current dispatch times of ~20μs seem unnecessarily high. Revisiting these algorithms may permit some performance improvement.
- Tune network performance. Although this is primarily an OSE-related shortcoming, the observed network behaviour was erratic.
- Fault-tolerance. Currently, GRRDE will cancel a subscription if the destination process or link fails. Further consideration is necessary to determine design guidelines for dealing with failure.
- Compartmentalize OS dependent features. GRRDE currently exploits the native commands of the OSE RTOS. In order to promote future portability, these OS dependencies should be isolated, and a clear plan for cross-platform development formulated.

11.3.2 Enhancements

Certain capabilities not currently provided by GRRDE could be added with further research. In contrast to refinements, enhancements would require substantial but reasonable modifications to the middleware. Examples include:

- Quality of Service. Subscription dispatching is not currently differentiated by the priority of the recipient. Lower latency and jitter for high-priority processes could be achieved if subscriptions were prioritized.
- Synchronization Primitives. Synchronization between modules is now largely done by hand. Built-in support would simplify the developer's tasks.
- Additional Services. Expand the GRRDE services to include other communications abstractions such as synchronous communication. RPC-like invocations could be added without extensive modifications to the current framework. Alternately, we might integrate a system like RT-CORBA to handle queries and other non-subscription exchanges. Other researchers have performed similar tasks and integrated CORBA with the SIMPLEX architecture [Polze, et al, 2000].

- Marshalling. One advantage of CORBA-type interactions is the automatic conversion of data types. Byte ordering and message structuring is all handled automatically at the interface between the user's code and the ORB. This is another source of developer workload, that could be reduced in a fairly structured manner.

11.3.3 Applications

Finally, we examine other places to use GRRDE. Specific suggestions for directly improving our application studies can be found in the preceding chapters. The following suggestions can guide the selection of new projects.

- Large development. Although the TechSat 21 simulation in Chapter 10 was quite large, it was not as complex nor as comprehensive as true flight software. An expanded development example, especially if it involves geographically separated contributors, can provide valuable feedback.
- Quantitative Studies. Attempts to quantify the improvement in productivity of using middleware for software development are difficult but not impossible. Such figures could be used to prioritize future improvements and development.
- Hardware-in-the-loop simulation. Although we have proposed strategies for migrating GRRDE simulations into deployed GRRDE flight software, the recommendations were hypothetical. Directly performing this task helps assess what further changes are necessary to the GRRDE framework, and strengthens the evidence for the utility of our middleware.

11.4 Wider GRRDE Applications

The bulk of this thesis has considered using GRRDE for distributed spacecraft software development. This focus should not be exclusive. Other applications can benefit from this tool as well. In this section we reflect on the particular system characteristics that we have addressed in the GRRDE middleware and examine other software engineering domains that possess these qualities.

Four properties are common to the space systems that we have considered: distribution, real-time criticality, control complexity, and system complexity. Distribution implies that we must provide communication between several processing devices. Criticality concerns

underscore the need for hard, deterministic, real-time guarantees. Additionally the software for these spacecraft also contains sophisticated, computationally-expensive feedback loops. These controllers must manage rapidly changing physical dynamics. Lastly, the subsystems are frequently tightly coupled, making the whole system complex. We designed GRRDE to aid the developer in dealing with these issues. The extent to which other domains display these same characteristics determines the benefit they can derive from the GRRDE services.

TABLE 11.1 Comparison of Real-Time Software Engineering Domains

Application	Distribution	Real-Time Criticality	Control Complexity	System Complexity	Benefit from GRRDE
Traditional Spacecraft	Low	High	Varies	Varies	Medium/Low
Advanced Spacecraft[a]	High	High	High	High	High
Aircraft	High	High	High	High	High
Train	High	High	Medium	Medium	Medium
Automobiles	Low	Low	Medium	Medium	Low
Mars Rover	High/Low[b]	Medium	Low	High	Low
Rail Switching	High	High	Low	High	Medium
Factory Automation[c]	Medium	Low	Low	Medium	Low
Factory Robotics	Medium	Medium	High	Medium	Medium
Factory Process Control	High	High	Medium	High	High
Nuclear Power Plants	High	High	Low	High	Medium

a. Such as those addressed in this study.
b. Depends on whether mission consists of single or multiple vehicles
c. e.g. a bottling plant

Table 11.1 is a qualitative assessment of the applicability of GRRDE to other real-time domains. Distribution and control complexity are particularly important in the determination of suitability. Their influence is both positive and negative. Domains with these characteristics are helped by GRRDE, but those without, may be better off *not* using our

middleware, since GRRDE may add development or run-time overhead. In contrast, addressing safety-criticality and system complexity when these features are not present, is not an inconvenience. Thus, software similar in character to that of advanced space systems will reap many of the same benefits from the GRRDE middleware. Several transportation and industrial applications are especially promising. Other applications may still benefit from the use of middleware, but should consider whether the particular services offered by GRRDE are best matched to the software responsibilities.

11.5 Final Word

Conventional flight software methods may be effective for near-term missions, but the tendency to rely on software to implement increasingly complex functions suggests limits to these techniques. As system complexity grows, the costs of software development and the likelihood of software failures will increase as well. Unless measures are taken to manage complexity and promote reliability, extensibility and scalability, flight software may limit mission capabilities rather than enable them. Middleware systems have been employed to good effect in terrestrial settings, and are now being adopted into embedded, real-time applications. Approaches such as those used in developing GRRDE, can be used to bring some terrestrial innovations into the spacecraft forum.

REFERENCES

[Anderson, et al, 1995] Anderson, T., Culler, D., Patterson, D., "The Case for Network of Workstations", *IEEE Micro*, Volume: 15 Issue: 1, Feb. 1995, Page(s): 54 -64

[Augustine, 1997] Augustine, N., *Augustine's Laws (6th Edition)*, American Institute of Aeronautics and Astronautics, December, 1997

[Bates, 1998] Bates, J., "The State of the Art in Distributed and Dependable Computing", *CaberNet Report*, Laboratory for Communications Engineering: University of Cambridge, 1998.

[Beichman, et al, 1999] Beichman, C., Woolf, N., Lindensmith, C., (eds.), *The Terrestrial Planet Finder: A NASA Origins Program to Search for Habitable Planets*, Jet Propulsion Laboratory Publication 99-3.

[Bergland, 1981] Bergland, G., "A Guided Tour of Program Design Methodologies", IEEE Computer, October 1981, Pages: 13-37

[Bernard, et al, 1998] Bernard, D., Dorais, G., Fry, C., Gamble Jr., E., Kanefsky, B., Kurien, J., Millar, W., Muscettola, N., Nayak, P., Pell, B., Rajan, K., Rouquette, N., Smith, B., Williams, B., "Design of the Remote Agent Experiment for Spacecraft Autonomy", Proceedings of IEEE Aerospace Conference, Snomass, CO, 1998

[Bernstein, et al, 1987] Bernstein, P., Hadzilacos, V., Goodman, N., *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987

[Bernstein, 1996] Bernstein, P., "Middleware: A Model for Distributed System Services", *Communications of the ACM*, Vol. 39, No. 2, February 1996, Page(s): 86-98

[Biesiadecki, et al, 1997] Biesiadecki, J., Jain, A., James, M., "Advanced Simulation Environment for Autonomous Spacecraft", *International Symposium on Artificial Intelligence, Robotics and Automation in Space* (i-SAIRAS'97), (Tokyo, Japan), Jul. 1997.

[Booton & Ramo, 1984] Booton Jr., R., Ramo, S., "The Development of Systems Engineering", IEEE Transactions on Aerospace and Electronic Systems, AES-20(4):306-309, July, 1984.

[Brown, 2001] Brown, G., "Radiation Hardened PowerPC 603E Based Single Board Computer", *Digital Avionics Systems, 2001. DASC. 20th Conference*, Volume: 2, 2001, Page(s): 8C1/1 -8C1/12 vol.2

[Bryant & Key, 1999] Bryant, S., Key, K., "Redefining the Process for Development of Embedded Software", *Proc. of the IEEE Intl. Symp. on Computer Aided Control Systems Design*, Kohala Coast, Hawaii, 1999, Pages: 261-266

[Burns & Wellings, 1996] Burns, A., and Wellings, A. *Real-Time Systems and Programming Languages*, Addison-Wesley, 1996

[Buttazzo, 1997] Buttazzo, G., *Hard Real-Time Computing Systems*, Kluwer Academic Press, 1997

[CDIO, 2001] Miller, D., et al, *ARGOS: Adaptive Reconnaissance Golay-3 Optical Satellite*, Critical Design Review Report, MIT Dept. of Aeronautics and Astronautics., 2001

[Chau, et al, 1995] Chau, S., Reh, K., Cox, B., Barfield, J., Lockhart, W., McLelland, M., "A Multi-Mission Space Avionics Architecture", *JPL Technical Report 95-1497*, 1995

[Clark, 1991] Clark, B., "Reconfigurable Software Development", *Digital Avionics Systems Conference, 1991. Proceedings., IEEE/AIAA 10th*, 1991, Page(s): 499 -503

[Coad & Yourdan, 1991] Coad, P.; and Yourdan, E.: *Object-Oriented Analysis.* Yourdan Press, Englewood Cliffs, N.J., 1991

[Costain, 1995] Costain, G., "A Comparison of CASE-Based O-O Methodologies Coad/ Yourdan OOA and Booch OOD", *Software Education Conference*, 1994. Proceedings, 1995, Page(s): 120 -127

[Crew, 2002] Personal conversations with Geoffery Crew, Research Scientist at the Center for Space Research (MIT) and flight software engineer for the HETE/HETE-2 spacecraft.

[Das & Cobb, 1998] Das, A.; Cobb, R.; "TechSat 21: Space Missions Using Collaborating Constellations of Satellites".SSC98-VI-1, *Proc. of the 12th An. AIAA/USU Conf. on Small Sat.* Logan, Utah, Sept. 1998

[Davis & Hamscher, 1988] Davis, R., Hanscher, W., "Model-Based Reasoning: Troubleshooting", *Exploring Artificial Intelligence*, Morgan Kaufman Publishers, 1988.

[Davis, 1984] Davis, R., "Diagnostic Reasoning Based on Structure and Behavior", *Artificial Intelligence*, 24:347-410, Dec. 1984

[de Kleer & Brown, 1992] de Kleer, J., Brown, J., "Model-Based Diagnosis in SOPHIE III", *Readings in Model-Based Diagnosis*, Hamscher, Console, de Kleer (eds.), Morgan Kaufman, 1992

[de Kleer & Williams, 1987] de Kleer, J., Williams, B., "Diagnosing Multiple Faults"

Artificial Intelligence, Vol. 32, No. 4, April 1987, Pages: 97-130.

[DeRemer & Kron, 1976] DeRemer, F., Kron, H., "Programming-In-the-Large Versus Programming-In-the-Small, IEEE Transactions On Software Engineering. 2(2):80-86, June 1976.

[Diersing, 1993] Diersing, R., "The Development of Low-Earth-Orbit Store-and-Forward Satellites in the Amateur Radio Service", 12th IEEE Conference on Computers and Communications, Pages: 378-386

[Doyle, 1998] Doyle, R., "Spacecraft Autonomy and Missions of Exploration", *IEEE Intelligent Systems*, Sept./Oct., 1998.

[Dvorak, 1992] Dvorak, D., "Monitoring and Diagnosis of Continuous Distributed Systems Using Semiquantitative Simulation." Ph.D. Dissertation, University of Texas at Austin, 1992

[Dvorak, et al, 1999] Dvorak, D., Rasmussen, R., Reeves, G., Sacks, A., "Software Architecture Themes in JPL's Mission Data System," *AIAA Space Technology Conference*, September 1999, AIAA-99-4553

[Dvorak, et al, 2000] Dvorak, D., Rasmussen, R., Reeves, G., Sacks, A., "Software Architecture Themes in JPL's Mission Data System," *2000 IEEE Aerospace Conference*, March 2000, Page(s). 259 - 267

[EC++, 2002] EC++ is an amendment to the C++ standard and can be found at the official website: http://www.caravan.net/ec2plus/

[Emmerich, 2000] Emmerich, W., "Software Engineering and Middleware: A Roadmap", *Proceedings of the (ACM) Conference on the Future of Software Engineering*, 2000, Limerick, Ireland., Page(s): 117-128

[Emmerich, 2000a] Emmerich, W., *Engineering Distributed Objects*, John Wiley and Sons, 2000

[Enright, et al, 1999] Enright, J., Sedwick, R., Miller, D., "Information Architecture Analysis and Optimization for Space-Based Distributed Radar", *AIAA Space Technology Conference*, Sept. 28-30, 1999, Albuquerque, New Mexico, (AIAA 99-4551).

[Erkkinen, 1999] Erkkinen, T., "Safety-Critical Software Generation", *Proc. of the IEEE Intl. Symp. on Computer Aided Control Systems Design*, Kohala Coast, Hawaii, 1999, Pages: 237-242

[Fay-Wolfe, et al, 2000] Fay-Wolfe, V., DiPoppo, L., Cooper, G., Johnston, R., Kortmann, P., Thuraisingham, B., "Real-Time CORBA", *IEEE Transactions on Parallel and*

Distributed Systems, Vol. 11, No. 10, October 2000, Page(s) 1073-1089.

[Fesq, 1993] Fesq, L., "MARPLE: An Autonomous Diagnostician for Isolating System Hardware Faults", Doctoral Dissertation, University of California: Los Angeles, 1993

[Fisher & Ghassemi, 1999] Fisher, S., Gassemi, K., "GPS IIF - The Next Generation", Proceedings of the IEEE, Vol. 87, No. 1, Jan. 1999, Pages: 24-47

[Garland & Lynch, 2000] Garland, S., Lynch, N., "Using I/O Automata for Developing Distributed Systems", in Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285-312, Cambridge University Press, 2000

[Garrison, et al, 1995] Garrison, T., Ince M., Pizzicaroli, Swan, P., "Iridium Constellation Dynamics: The Systems Engineering Trades", 46th International Astronautical Conference, Oct. 4-6, 1995, Oslo, Norway, IAF-95-U.2.04

[Geist et al 1996] Geist, A.; Beguelin, A.; Dongarra, J.; Jiang, W.; Manchek, R.; Sunderam, V. *PVM Parallel Virtual Machine* MIT Press, 1996

[Goldman, 2001] Goldman, J., "X-38 Touts Ultimate Wireless OS", *TechTV*, December 10th, 2001, website: http://www.techtv.com/news/culture/story/0,24195,3364321,00.html

[Hall, 1996] Building Client/Server Applications Using Tuxedo, Wiley, 1996

[Hapner, et al, 2001] Hapner, M., Burridge, R., Sharma., R., Fialli, J., *Java Message Service Specification*, Technical Report, http://java.sun.com/products/jms, Aug. 2001

[Harrison, et al, 1997] Harrison, T., Levine, D., Schmidt, D., "The Design and Performance of a Real-Time CORBA Event Service", *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications,* October 1997, Volume 32 Issue 10, Page(s): 184-200

[Hatton, 1998] Hatton, L., "Does OO Synch with How We Think?", *IEEE Software*, May/June 1998.

[Heitmeyer & Mandrioli,1996] Heitmeyer, C. Mandrioli, D. (eds.), Formal Methods for Real-Time Computing, John Wiley Press, 1996

[Hortsmann & Kirtland, 1997] Hortsmann, M. and Kirtland, M. "DCOM Architecture", Microsoft Corporation, MSDN Library, 1997

[Hudders, 1994] Hudders, E., *CICS: A Guide to Internal Structure*, Wiley, 1994.

[ISO/IEC, 1994] ISO/IEC 7498-1, *Information technology -- Open Systems Interconnection -- Basic Reference Model: The Basic Model,* 1994, International Standards Organization.

[ISO/IEC, 1996] ISO-20746-2, *Information Technology - Open Distributed Processing-Reference Model: Foundations*, 1996, International Standards Organization.

[ISO/IEC, 1998] ISO/IEC 14882, *Programming languages -- C++*, 1998, International Standards Organization.

[Jain & Man, 1992] Jain, A., Man, G., "Real-Time Simulation of the Cassini Spacecraft using DARTS: Functional Capabilities and the Spatial Algebra Algorithm", *5th Annual Conference on Aerospace Computational Control*, Aug., 1992

[Jones, et al, 1998] Jones, M. K., Carrion, C., Klassen, E. L., "SUPERMOCA: Commercially Derived Standards for Space Mission Monitor and Control", *AIAA Defense and Civil Space Programs Conference and Exhibit*, Huntsville, AL, Oct. 28-30, 1998(A98-45901 12-66), Page(s) 29-38

[Kolcio, et al, 1999] Kolcio Ksenia O., Hanson, Mark L., Fesq, Lorraine M., Forrest, David J. "Integrating Autonomous Fault Management With Conventional Flight Software: A Case Study", *Aerospace Conference, 1999. Proceedings. 1999 IEEE*, Volume: 1, 1999, Page(s): 307 -314 vol.1

[Kopetz, 1997] Kopetz, H., *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Press 1997

[Kopetz, 2000] Kopetz, H., "Software Engineering for Real-Time: A Roadmap", *Proceedings of the (ACM) Conference on the Future of Software Engineering*, 2000, Limerick, Ireland., Page(s): 201-211

[Leang, et al, 1997] Leang, C., McMahon II, E., Pingree, P., Basilo, R., "Real-Time Testbed Spacecraft Simulation for the Deep Space One Spacecraft", *Digital Avionics Systems Conference*, 1997. 16th DASC., AIAA/IEEE, Volume: 2, 1997, Page(s): 8.2 -15-8.2-21 vol.2.

[Lee, 2000] Lee, E., "What's Ahead for Embedded Software?", *IEEE Computer Magazine*, September 2000, Page(s): 18-26

[Lehoczky, et al, 1989] Lehoczky, J., Sha, L., Ding, Y., "The Rate Monotonic Scheduling Algorithm - Exact Characterization and Average Case Behavior", *IEEE Real-Time Systems Symposium*, Dec. 1989

[Lehoczky & Ramos-Thule, 1992] Lehoczky, J., Ramos-Thule, S., "An Optimal Algorithm for Scheduling Soft-aperiodic Tasks in Fixed Priority Preemptive Systems", *13th IEEE Real-Time Systems Symposium*, Pages: 110-123, 1992

[Leitner, 2001] Leitner, J., "A Hardware-in-the-Loop Testbed for Spacecraft Formation Flying Applications", *IEEE Aerospace Conference*, 2001, Vol. 2, Pages: 2/615-2/620

[Leveson, 1992] Leveson, N., "High-Pressure Steam Engines and Computer Software", *IEEE Computer*, October 1994

[Leveson 1995] Leveson, N., *Safeware: System Safety and Computers*, New York: Addison-Wesley 1995

[Leveson, 2000] Leveson, N., "Completeness in Formal Specification Language Design for Process Control Systems" *Proceedings of Formal Methods in Software Practice Conference*, August 2000

[Leveson, 2001] Leveson, N., "Systemic Factors in Software-Related Spacecraft Accidents", AIAA Space 2001 Conference and Exposition, Albuquerque, NM, Aug. 28-30, 2001, AIAA 2001-4763.

[Liljedahl & Lillieskold, 1999] Liljedahl, O., Lillieskold, C., *OSE Performance Measurements R1.1*, ENEA Corporation White Paper, 1999-05-12

[Lions, 1996] Lions, J.L. (chair) "Ariane 5: Flight 501 Failure", *Report of the Inquiry Board*, Paris, July 19, 1996.

[Liu & Layland, 1973] Liu, C. L., and Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment," *Journal of the Association for Computing Machinery* 20, 46-61, January 1973

[Locke, 1992] Locke, C., "Software Architecture for Real-Time Applications: Cyclic Executives versus Fixed Priority Executives", *Real-Time Systems*, 4(1), Page(s):37-53.

[Lynch, 1996] Lynch, N.A., *Distributed Algorithms*, Morgan Kaufman Publishers, Inc., CA, 1996.

[Lutz, 1992] Lutz, W., "Analyzing Software Requirements Errors in Safety-Critical Embedded Systems", I*EEE International Symposium on Requirements Engineering*, 1993, Page(s): 126 - 133.

[Maclay, 2000] Maclay, D., "Click and Code", *IEE Review*, May 2000, Pages: 25-28

[Malcom & Utterback, 1999] Malcom, H., Utterback, H., "Flight Software in the Space Department: A Look at the Past and a View Toward the Future", Johns Hopkins APL Technical Digest, Vol. 20 No. 4, 1999, Pages: 522-532.

[Marais, 2001] Marais, K., "The Development and Analysis of Scanned Pattern Interferometric Radar", Dept. of Aeronautics and Astronautics (MIT), Masters Thesis,

and MIT-Space Systems Laboratory publication (SSL #9-01), 2001

[Marr, 1994] Marr IV, J., "Performing the Galileo Mission Using the S-Band Low-Gain Antenna", *IEEE Aerospace Applications Conference*, 1994, Page(s): 145 -183

[Marti, et al, 2001] Marti, P., Fuertes, J., Fohler, G., Ramamritham, K., "Jitter Compensation for Real-Time Control Systems", *22nd IEEE Real-Time Systems Symposium*, 2001. Page(s): 39 -48

[Marshall, 1981] Marshall, M. "Goals for the Air Force Autonomous Spacecraft" *USAF Report-SD-TR-81-72*, JPL Report 7030-1

[Miller, et al, 2001] Miller, D., Sedwick, R., Hartman, K., "Evolutionary Growth of Mission Capability Using Distributed Sparse Apertures", *Internal Report*, Space Systems Laboratory, MIT.

[Mills, 1998] Mills, M. "A Call from the Heavens Above", *Washington Post,* Nov. 23, 1998, Page: F23.

[Motorola, 2001] Motorola Corporation, *MPC750 RISC Microprocessor Family User's Manual*, Motorola Product Specification, MPC750UM/D, Dec. 2001.

[Muirhead, 1997] Muirhead, B., "Mars Pathfinder Flight System Integration and Test",

[Nordwall, 1993] Nordwall, B., "McDonnell Douglas Aviation Engineers", *Aviation Week and Space Technology*, Vol. 139., No.11, Page: 94, Sept. 13, 1993

[OMG, 2000] The Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Rev 2.4.1 November 2000, Page(s) 871-927

[OMG, 2001] The Object Management Group, Event Service Specification, V 1.1, March 2001.

[Orehek & Robl, 2001] Orehek, M., Robl, C., "Model-Based Design of an ECU with Data- and Event-Driven Parts Using Auto Code Generation", Proc. IEEE International Conference on Robotics and Automation, Seoul, Korea, May 21-26, 2001, Page(s): 1346-1351.

[Parnas, 1972] Parnas, D., "A Technique for Software Module Specification with Examples", *Communications of the ACM*, Vol. 15, No.5, May 1972, Pages: 330-336.

[Polze, et al, 2000] Polze, A., Schwarz, J., Wehner, K., Sha, L. "Integration of CORBA Services with a Dynamic Real-Time Architecture", 6th IEEE Real-Time Technology and Applications Symposium, 2000, Page(s): 198-206

[Ptak & Foundy, 1998] Ptak, A., Foundy, K., "Real-Time Spacecraft Simulation and Hardware-in-the-Loop Testing," *Fourth IEEE Real-Time Technology and Appli-*

cations Symposium, 1998., Page(s): 230 -236

[Rajkumar, et al, 1995] Rajkumar, R., Gagliardi, M., Sha, L., "The Real-Time Publish/ Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation", First IEEE Real-Time Technology and Applications Symposium, May 1995, Page(s): 66-75.

[Rayman, et al, 1999] Rayman, M., Varghese, P., Lehman, D., Livesay, L., "Results from the Deep Space 1 Technology Validation Mission", *50th International Astronautical Congress*, Amsterdam, The Netherlands, IAA-99-IAA-11.2.01

[Robertson, et al, 2000] Robertson, B., Sabelhaus, P., Mendenhall, T., Fesq, L., "The Recovery of TOMS-EP", Advances in the Astronautical Sciences, American Astronautical Society, Vol. 104, AAS-00-076, Pages 665-685

[Sary & Werking 1997] Sary, C.; Werking, C.; *Intelligent Systems Applied to the Aerospace Industry* Proc. of the 11th An. AIAA/USU Conf. on Small Sat. Logan, Utah, Sept. 1997

[Seto, et al, 1998] Seto, D., Krogh, B., Sha, L., Chutinan, A., "The Simplex Architecture for Safe On-Line Control System Upgrades", *Proc. of the American Control Conference*, June 1998, Page(s): 3504-3508

[Sha, et al, 1994] Sha, L., Rajkumar, R., Sathaye, S., "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Distributed Systems", *Proceedings of the IEEE*, Vol. 82 No. 1, January 1994.

[Shaw, 1998] Shaw, G. B. *The Generalized Information Network Analysis Methodology for Distributed Satellite Systems.* Ph.D. Thesis, MIT Dept. of Aeronautics and Astronautics. 1998

[Shaw, et al, 1999] Shaw, M., Levin, P., Martel, J., "The DoD: Stewards of the Global Information Resource, the Navstar Global Positioning System", Proceedings of the IEEE, Vol. 87, No. 1, January 1999, Pages: 16-23

[Shin & Chang, 1995] Shin, K., Chang, Y., "A Reservation-Based Algorithm for Scheduling both Periodic and Aperiodic Real-Time Tasks", *IEEE Transactions on Computers*, Vol. 44, No. 12, Dec. 1995, Pages: 1405-1419

[Shore, 1986] Shore, J., *The Sachertorte Algorithm and Other Antidotes to Computer Anxiety*, Penguin Books, New York, 1986

[Schmidt, et al, 1997] Schmidt, D., Levine, D., Mungee, S., "The Design of the TAO Real-Time Object Request Broker," *Computer Comm. J.*,Summer 1997.

[Schweighart & Sedwick, 2002] Schweighart, S., Sedwick, R., "A High-Fidelity Linear-

ized J2 Model for Satellite Formation Flying", *AIAA Journal of Guidance and Control, forthcoming.*

[Smith, 1999] Smith, B (ed.), "TOMS Back On-Line", *Aviation Week and Space Technology*, Vol. 250, No. 5, Page. 23. February, 1, 1999

[Smith & Elbs, 1999] Smith, M., Elbs, M., "Towards a More Efficient Approach to Automotive Embedded Control System Development", *Proc. of the IEEE Intl. Symp. on Computer Aided Control Systems Design*, Kohala Coast, Hawaii, 1999, Pages: 219-224

[Snir, et al, 1995] M. Snir, S.W. Otto, S. Huss-Lederman, D. W. Walker and J. J. Dongarra, *MPI: The Complete Reference*, MIT Press, 1995.

[Spector & Gifford, 1984] Spector, A., Gifford, D., "The Space Shuttle Primary Computer System", Communications of the ACM, Vol. 27, No. 9, September 1984, Pages: 872-900

[Speight & Bennett, 1998] Speight, E.; Bennett, J.K., "Using multicast and multithreading to reduce communication in software DSM systems", *IEEE High-Performance Computer Architecture, 1998. Proceedings.*, Page(s): 312 -322

[Stewart, 1999] Stewart, D., "30 Pitfalls for Real-Time Software Developers, Part 1", *Embedded Systems Programming,* October 1999, Page(s): 32-41.

[Stewart, 1999a] Stewart, D., "More Pitfalls for Real-Time Software Developers", *Embedded Systems Programming*, November, 1999, Page(s): 74-86.

[Stolper, 1999] Stolper, S., "Streamlined Design Approach Lands Mars Pathfinder", IEEE Software, Sept./Oct. 1999. Pages: 52-61

[Strosnider, et al, 1995] Strosnider, J., Lehoczky, J., Sha, L., "The Deferrable Server Algorithm for Enhancing Aperiodic Responsiveness in Hard Real-Time Environments", IEEE Transactions on Computers, Vol. 44, No. 1, January 1995, Pages: 73-91.

[Sun, 1988] Sun Microsystems, "RFC 1057: Remote Procedure Call Protocol specification Version 2", Internet Engineering Task Force. Jun.-01-1988.

[Surka, et al, 2001] Surka, D., Brito, M., Harvey, C., "The Real-Time ObjectAgent Flight Software Architecture for Distributed Satellite Systems", *IEEE Aerospace Conference*, Volume: 6, 2001, Page(s): 2731 -2741.

[Tanenbaum, 1976] Tanenbaum, A., "In Defense of Program Testing or Correctness Proofs Considered Harmful", *SIGPLAN Notices*, Vol. 11 No. 5.

[TRW, 1994] TRW Corporation, "ROCSAT-1 Software Standards and Procedures Man-

ual", internal document D22847, December 1994.

[Twiggs, et al, 1999] Twiggs, R., Cutler, J., Hutchinson, G., Williams, J., "OPAL - A First Generation Microsatellite that Provides Picosat Communications for the Amateur Radio Community", *17th AMSAT-NA Space Symposium*, A00-29732 07-32, 1999, Pages: 40-47

[Verissimo & Rodrigues, 2001] Verissimo, P., Rodrigues, L., *Distributed Systems for System Architects*, Kluwer Academic Publishers, 2001

[Vitruvius, 1960] Vitruvius, P., The Ten Books on Architecture (written 27BC), translated by M.H. Morgan, Dover Publications, New York.

[Vytopil, 1993] Vytopil, J. (ed.), *Formal Techniques in Real-Time and Fault-Tolerant Software*, Kluwer Academic Publishers, 1993.

[Wagner, 1998] Wagner, D., "Spacebourne Processors: Past, Present and Future Satellite Onboard Computers," *49th International Astronautical Congress*, Sept. 28-Oct. 2. 1998, Melbourne, Australia, ST-98-W.1.01,

[Wie & Barba, 1984] Wie, B., Barba, P., "Quaternion Feedback for Spacecraft Large Angle Maneuvers", Journal of Guidance, Control and Dynamics, Vol. 8 No. 3, 1984.

[Williams & Nayak, 1996] WIlliams, B., Nayak, P., "Immobile Robots: AI in the New Millennium", *AI Magazine*, Fall 1996.

[Williams, et al, 2002] Williams, B., Ingham, M., Chung, S., Elliot, P., "Model-based Programming of Intelligent Embedded Systems and Robotic Explorers", submitted to *IEEE Proceedings Special Issue on Embedded Software*, Jan. 2002.

[Wright & Williams, 1993] Wright, D.T.; Williams, D.J., "Object-like Software Design Methods for Intelligent Real-time Process Control," *Intelligent Control, 1993., Proceedings of the 1993 IEEE International Symposium on*, 1993, Page(s): 144 - 149-+

[Zita Haigh, et al, 2000] Zita Haigh, K., Musliner, D. J., Ghosh, S., "RT-MLab: Really Real-Time Robotics" *AAAI Spring Symposium on Real-time Autonomous Systems*, Spring 2000

Appendix A

SECONDARY TOOLS

The subscription services offered by the GRRDE Publish Subscribe System represent the primary implementation aids in GRRDE. The framework also provides a number of other tools and services that automate common or tedious tasks.

Simulation Aids. One of most commonly used simulation tools is a generic numerical propagator class. For integration, it uses the sixth-order Runge-Kutta method with variable step size and defect correction. An advanced interpolator ensures that interpolated values have tolerances comparable to the endpoints. This can save processor time. Other tools include linear time-independent (LTI) filters and advanced random number generation.

Mathematical Tools. To aid GRRDE development the SIGLIB mathematical package has been purchased. It is a set of linear algebra and signal processing routines optimized for embedded applications.

Synchronization Tools. Real-time systems frequently require control over the synchronization of periodic processes. GRRDE provides several convenient mechanisms for enforcing timing and starting constraints between different blocks and processes.

Atomic Objects. In systems where an object or variable may be accessed concurrently by multiple processes, it is important that behaviour remain consistent or *atomic*. Any invo-

cation on an object, such as a *read* or *write*, takes a finite amount of time. Since there are no restrictions on when an external invocation may come, it is possible that these periods may overlap. Atomicity is defined to be a property that ensures that a total ordering of momentary operations can be found that reflects the observed extended behaviours. For example consider an object A consisting of two integers. The initial value is $A = \{1, 2\}$. Two extended operations, a $readA(\ )$ and a $writeA(3, 4)$ arrive together. The object is atomic if it can be guaranteed that the final state is $A = \{3, 4\}$ and the *readA* operation returns either $\{1, 2\}$ *or* $\{3, 4\}$. GRRDE provides a general class of atomic object data types that implement the Emulated Priority Ceiling Protocol [Sha, et al, 1994] to avoid priority inversion effects.

Interface Tools. While automated testing requires little interaction, full operational simulations need convenient means of user interaction. GRRDE provides interface tools that allow communication between the OSE-based simulation environment and standard Windows-based graphic user interfaces. This interface can be used to represent both ground operator activities as well as simulation steering. Simple visualization utilizing the OpenGL three-dimensional graphics libraries are included.

Configuration Management. One of the benefits that GRRDE offers is the capability to rapidly configure a simulation. A simulation configuration consists of lists of modules to load, where to load them and any required initialization parameters. These setup files are parsed automatically upon system start. A simple relational database was assembled using Microsoft Access to manage interface definitions, module revisions and system configurations.

Appendix B

INTERFACE DEFINITION CONVENTIONS

This appendix presents an example of the interface documentation standards used for GRRDE module design. The structure of the specifications is described in Section 4.4. Our chosen example corresponds to the orbit and attitude propagator module of the TechSat 21 simulation of Chapter 10. Not only do they provide concrete illustrations of the specification convention, they represent a convenient reference document for the included tools. The visual representation of the module specification has not been standardized. Recommended guidelines for presentation may be developed in the future; until that time, users are encouraged to adopt any convenient and easily understood formatting.

B.1 Orbit/Attitude Propagator

This module is a dynamics simulator for the spacecraft in the TechSat 21 simulation. It is responsible for integrating the non-linear state equations for both spacecraft orbit and attitude.

SN-5 Service Name: *gflops_obt_propUID#.* Provides orbit propagation information for each satellite in system

Interface Definition Filename. gflops_orbit.sig

DPN-5.1 Data Product Name: *obt_PosVel.* Return satellite state in terms of Earth-Centred Inertial Cartesian position and velocity vectors

Type. Time-Triggered Output

Signal Number. 60703 (ORBIT_STATE_SIG)

Structure. This structure is shared by a number of DPNs

```
struct OrbitStateSig{
SIGSELECT sigNo;
int iStateType;
intiSCId; // S/C id #
double dX[6]; // State vector
double dTimeStamp; //J2000 date (in days)
};
```

Since the data structure is shared by a number of state representations, the value of iStateType determines how the data should be interpreted. The allowable values are detailed in Table B.1.

TABLE B.1 State Vector Types

iStateType	Meaning
1	Position (m)[a]
2	Position (m), Velocity (m/s)
3	Geodetic: Lat[b], Long, Altitude (m)
4	Equinoctial Elements: a (m), P1, P2, Q1, Q2, M
5	Classic Elements: a (m), e, i, ω, Ω, M

a. Uses only first three elements of X

b. All angular measures are in radians

Position and velocity are the in Earth-Centred inertial reference frame. The x-axis is defined by the direction of the vernal equinox, the z-axis by the Earth's spin axis, and the y-axis results from the requirements that the three principal axes form a dextral (right-handed) reference frame.

Period. 1 ms

ARGC. in the current design, all spacecraft elements are propagated by the same module. The value of argc must be set to the UID of the desired spacecraft. The standard range is from 0-7.

ARGV. This parameter is ignored

DPN-5.2 Data Product Name: *obt_Equin.* Equinoctial elements of a satellite.

Type. Time-Triggered Output

Signal Number. 60703 (ORBIT_STATE_SIG)

Structure. See DPN-5.1. This data product uses the same data structure with a corresponding change in the state type. Although not as intuitive as the classic Keplerian orbital elements, the so-called equinoctial elements are better behaved mathematically. With the exception of completely retrograde orbits, this representation is singularity free.

Period. 1 ms

ARGC. in the current design, all spacecraft elements are propagated by the same module. The value of argc must be set to the UID of the desired spacecraft. The standard range is from 0-7.

ARGV. This parameter is ignored

DPN-5.3 Data Product Name: *obt_Geodetic.* Satellite position in geodetic coordinates.

Type. Time-Triggered Output

Signal Number. 60703 (ORBIT_STATE_SIG)

Structure. See DPN-5.1. This data product uses the same data structure with a corresponding change in the state type. Latitude Range is $-\frac{\pi}{2}$[South]...$\frac{\pi}{2}$[North], Longitude range is $-\pi$[West]...π[East]. Altitude is measured in m and is corrected for oblateness.

Note that geodetic latitude is measured from the equatorial plane to the surface normal (Figure B.1). The Greenwich Apparent Siderial Time (GAST) is estimated with a simple model of nutation and precession .

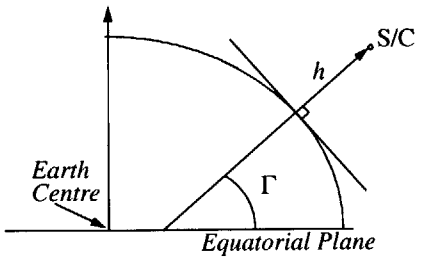


Figure B.1 Definition of Geodetic Latitude (Γ) and altitude (*h*).

Period. 1 ms

ARGC. in the current design, all spacecraft elements are propagated by the same module. The value of argc must be set to the UID of the desired spacecraft. The standard range is from 0-7.

ARGV. This parameter is ignored

DPN-5.4 Data Product Name: *Orbit_Set_Elements.* Initializes orbital elements

Type. Command Input

Signal Number. 60704 (ORBIT_SET_ELEMENTS)

Structure. Elements are set using the following structure:

```
struct OrbitSetElements{
SIGSELECT sigNo;
int iStateType;
intiSCId;
bool bIdeal;
double dX[6];
};
```

The state type is specified using the same constants defined in DPN-5.1. Only types 4 (Equinoctial) and 5 (Classic) are supported. The bIdeal field specifies the disturbance model. If this field is set to *true*, no disturbances will be included in the propagator. If set to false the full fidelity simulation will be used. The full simulator includes oblateness effects (J2 through J4) as well as Sun and Moon influence. Settings apply to the spacecraft specified in the iSCId field. Each spacecraft may select a propagator model independently. If this signal is received before the propagator is started (see DPN-5.5), the setting applies to the start of the simulation. If the signal is received after the simulation has begun, it will apply instantly.

Period. N/A

ARGC. N/A

ARGV. N/A

DPN-5.5 Data Product Name: *Orbit_Start_Prop.* Starts the orbital propagator.

Type. Command Input

Signal Number. 60705 (ORBIT_START_PROP)

Structure. There is no payload to this command, it is just a message tag. After receiving this signal the, propagator will begin simulating spacecraft orbits.

Period. N/A

ARGC. N/A.

ARGV. N/A

DPN-5.6 Data Product Name: *Force_Impulse.* Applies impulsive velocity change to specified spacecraft.

Type. Command Input

Signal Number. 60718 (FORCE_IMPULSE_SIG)

Structure. The following structure describes the impulse:

```
struct ForceImpulseSig {
SIGSELECT sigNo;
int iSCId;
double dImpulse[3];
};
```

The impulse is actually an instantaneous velocity change (m/s). It is applied in the orbit frame. The x-axis is defined by the radius vector, the z-axis is aligned with the angular velocity vector, and the y-axis is given by the appropriate cross-product. ($\hat{y} = \hat{z} \times \hat{x}$)

Period. N/A

ARGC. N/A

ARGV. N/A

SN-6 Service Number: *gflops_att_propUID#.* This service is concerned with providing the propagated spacecraft attitude.

Interface Definition Filename. gflops_orbit.sig

DPN-6.1 Data Product Name: *obt_AttQuat.* Provides quaternion representation of spacecraft attitude.

Type. Time-Triggered Output

Signal Number. 60708 (ATT_STATE_SIG)

Structure. The following structure describes the spacecraft attitude:

```
struct AttStateSig {
SIGSELECT sigNo;
int iStateType;
```

```
int iSCId;
double dQ[4];
double dTimeStamp; // J2000 time (in days).
};
```

This structure is shared with DPN-6.3. The iStateType determines the meaning to the quaternion. A value of ATT_STATE_INERTIAL is used to provide the rotation from inertial coordinates. The UID for the spacecraft is given in the iSCId field. The current valid range is 0-7. The first element of the quaternion is the η quantity while the last three elements form the $\hat{\varepsilon}$ vector. The quaternion represents the transform from the Earth-Centred Inertial coordinate frame to the spacecraft principal axes.

Period. 1 ms.

ARGC. In the current design, all spacecraft attitudes are propagated by the same module. The value of argc must be set to the UID of the desired spacecraft. The standard range is from 0-7.

ARGV. N/A

DPN-6.2 Data Product Name: *orb_AttFull.* Returns the full attitude state consisting of a quaternion representation of attitude as well as angular velocity.

Type. Time-Triggered Output

Signal Number. 60709 (ATT_FULL_STATE)

Structure. The following structure defines full attitude state signal.

```
struct AttFullState {
SIGSELECT sigNo;
int iSCId;
double dX[7];
double dTimeStamp; //J2000 time (in days)
};
```

The first four elements of the dX field contain the quaternion attitude representation described in DPN-6.1.The last three elements make up the angular velocity vector ω. It

has units or rad/s and describes the angular velocity of the spacecraft in the *body* frame. The spacecraft UID is provided in the iSCId.

Period. 1 ms.

ARGC. In the current design, all spacecraft attitudes are propagated by the same module. The value of argc must be set to the UID of the desired spacecraft. The standard range is from 0-7.

ARGV. N/A

DPN-6.3 Data Product Name: *orb_AttOrbitFrame.* Returns the quaternion representation of the spacecraft attitude in the orbital frame.

Type. Time-Triggered Output

Signal Number. 60708 (ATT_STATE_SIG)

Structure. This signal shares its message structure with DPN-6.1. The UID for the spacecraft is given in the iSCId field. The current valid range is 0-7. The first element of the quaternion is the η quantity while the last three elements form the $\vec{\varepsilon}$ vector. The field iStateType is assigned a value of ATT_STATE_ORBIT for quaternions referenced to the orbit reference frame. The quaternion represents a rotation from the spacecraft orbit frame of *reference.* The x-axis is defined by the radius vector, the z-axis is aligned with the angular velocity vector, and the y-axis is given by the appropriate cross-product. ($\hat{y} = \hat{z} \times \hat{x}$)

Period. 1 ms.

ARGC. In the current design, all spacecraft attitudes are propagated by the same module. The value of argc must be set to the UID of the desired spacecraft. The standard range is from 0-7.

ARGV. N/A

DPN-6.4 Data Product Name: *Impulse_Torque.* Apply an impulsive torque to the spacecraft.

Type. Command Input

Signal Number. 60719 (TORQUE_IMPULSE_SIG)

Structure. The impulsive torque signal structure is define as:

```
struct TorqueImpulseSig {
SIGSELECT sigNo;
int iSCId;
double dImpulse[3];
};
```

In contrast to the translational impulse (DPN-5.6), which is actually an impulsive velocity change (independent of mass), the impulsive torque represents a change in angular momentum. It has units of $kg \cdot m^2/s$.

Period. N/A

ARGC. N/A

ARGV. N/A

DPN-6.5 Data Product Name: *Set_Attitude.* Initialize a spacecraft's attitude state

Type. Command Input

Signal Number. 60707 (ATT_SET_STATE)

Structure. The spacecraft initialization uses the following structure.

```
struct AttSetState {
SIGSELECT sigNo;
int iSCId;
double dX[7];
};
```

This command sets the initial spacecraft attitude. The iSCId field selects the desired spacecraft and the dX entry contains the desired state (quaternion and angular velocity). Users should also set the spacecraft moments of inertia before starting the simulation. As with the orbit related functions, a set attitude command specifies the attitude when the simulation begins. In a running simulation, setting the attitude will cause an instantaneous change. Subsequent signals sent for the same spacecraft will override and previous settings.

Period. N/A

ARGC. N/A

ARGV. N/A