Order Number 9308362

Generation of synthetic workloads for distributed real-time computing systems

Kiskis, Daniel Lee, Ph.D.

The University of Michigan, 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

GENERATION OF SYNTHETIC WORKLOADS FOR DISTRIBUTED REAL-TIME COMPUTING SYSTEMS

by

Daniel Lee Kiskis

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1992

Doctoral Committee:

Professor Kang G. Shin, Chair
Assistant Professor Atul Prakash
Assistant Professor Chinya V. Ravishankar
Assistant Professor Stuart Sechrest
Professor Daniel Teichroew

RULES REGARDING THE USE OF

MICROFILMED DISSERTATIONS

Microfilmed or bound copies of doctoral dissertations submitted to The University of Michigan and made available through University Microfilms International or The University of Michigan are open for inspection, but they are to be used only with due regard for the rights of the author. Extensive copying of the dissertation or publication of material in excess of standard copyright limits, whether or not the dissertation has been copyrighted, must have been approved by the author as well as by the Dean of the Graduate School. Proper credit must be given to the author if any material from the dissertation is used in subsequent written or published work.

To my parents

ACKNOWLEDGEMENTS

First, I must thank Prof. Kang Shin for believing in me from the start and for supporting and encouraging me since. He gave me the freedom to follow my own path and gave me direction and kept me focused when I needed it. I could not have hoped for a more dedicated or more caring advisor. I would like to thank the remainder of my doctoral committee, Professors Atul Prakash, Chinya Ravishankar, Stuart Sechrest, and Daniel Teichroew for their constructive criticisms on this dissertation.

I would like to thank the National Aeronautics and Space Administration for supporting me under grant NAG-1-496.

For their positive influences on this dissertation and its author, I want to thank the following people. Prof. John Meyer for being an excellent teacher and for showing me the value of precision, accuracy, and rigor in performance evaluation. Prof. Michael Walker and Joseph Dionese for providing me with the source code for the robot controller and for showing me how to make the robot arm move.

Dilip Kandlur for letting me work on the early version of HARTOS, thus instilling in me an appreciation of the need for synthetic workloads to evaluate such systems. Paul Dodd for developing HMON, a real-time monitor, without which the experiments in Chapter 7 could not have been performed. Lup-Houh Ng for help in collecting the library of synthetic operations. Dave Musliner for proofreading this dissertation and providing many useful criticisms, and for developing *go* and *bibdb*, two tools that made the writing of this dissertation much easier. Jim Dolter for technical discussions and for keeping us supplied with the most up-to-date software possible. The past and present members of the Real-Time Computing Laboratory for encouragement and insight. B. J. Monaghan for taking care of our paperwork and for keeping us supplied with candy, coffee, and friendship.

Marty Stytz for getting me involved in CSEG and for being around when I needed to blow off during the early years of my graduate work. Phil MacKenzie for being there when I needed to wing some 'bee. Tom Makowski and Brad Bruno, my former roommates, for making sure I came home from work occasionally.

Gary Clark and Tim Householder, my long-time friends, for helping me through the hard times and helping me celebrate the good times. Rajalakshmi Subramanian for her

constant support, encouragement, and patience, and for making it all worthwhile. Finally, and most importantly, I want to thank my family. They gave me the strength to follow my dreams. Without them, this dissertation would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

Appendix

CHAPTER 1

INTRODUCTION

All computer systems have performance requirements that must be met. However, real-time systems have performance constraints that are especially stringent compared to those for non-real-time systems. Real-time applications include controlling such critical systems as nuclear reactors and aircraft. The time-critical nature of these applications demands that the real-time computer system provide a sufficiently high level of performance. The ability of the system to provide this performance is determined through evaluation, which may take a number of forms: analytic, simulation, or experimentation. Analytic and simulation evaluations are important steps in the initial design of a system, because they provide valuable estimates of its performance. Although they can be used to evaluate various design issues, these techniques have limitations. Due to tractability constraints, they use approximations that limit their accuracy. The actual performance can only be measured through experimental evaluation of the target system. To experimentally evaluate a system, it must be measured while executing an appropriate workload.

A *workload* is the collection of user inputs into a system. It consists of a set of tasks, the tasks' input data, and user commands. A workload produces demands for the system's resources; these demands are the *workload characteristics*. The performance of a system is a function of its workload. The structure and behavior of the workload directly affect the values of the performance indices that are measured during experiments. Therefore, an understanding of the workload is essential if any meaning is to be placed on the evaluation of the system.

There are two types of experimental evaluation. The first type is aimed at determining the performance under expected operating conditions. The target system is measured while executing a workload which is representative of the system's actual or proposed application workload. For this case, the actual application may be used as the workload, if it is available. The second type of evaluation is aimed at characterizing the performance of the system as a function of selected workload characteristics. For this type of evaluation, the

workload must exercise certain system components in user-specified ways. These workloads are usually custom programs designed to produce specific demands on system resources.

A *synthetic workload* (SW) is useful for both types of evaluation. An SW is an executable model of an actual workload. An SW reproduces the resource demands of the workload at a user-specified level of abstraction. It is composed of a set of parameterized synthetic tasks that are designed to use specific system resources. The synthetic tasks are constructed using operations that represent the types of operations performed by the tasks in the real workload, but they do not necessarily reproduce the exact sequence of operations, nor do they process real data. Instead, they perform operations on fixed-valued or randomly-generated data. For example, to model a floating-point intensive, signal-processing task in a real workload, the SW might execute a number of floating point arithmetic operations on a fixed data set. These operations will exercise the appropriate system hardware, say the floating point coprocessor, to the same degree as the real workload. However, the SW requires no signal data to process, and the code for the synthetic task may consist of a simple loop containing a short sequence of floating point operations. To reproduce the behavior of the signal-processing task, the loop is executed a number of times until the appropriate number of operations is executed. The user does not have to produce correct input data streams or complex algorithms to process the data. The user controls the workload characteristics by adjusting the parameters of the task. SWs are useful for experimental evaluations because they are flexible, their behavior is controllable and reproducible, and they are generally more compact than real workloads.

For the first type of evaluation, the SW can be used when the actual application software is unavailable, as is frequently the case when the target system is new or experimental. For a new system, the software may still be in the design stage at the time when the hardware and system software are ready to be evaluated. For experimental systems, a specific application may not yet be determined for the system. In these cases, the SW can be constructed based on the high-level requirements specification of the proposed workload or based on a characterization of the types of workloads which compose the application domain for the system. Even if the application workload is available, the SW may be preferable because its behavior is reproducible. The behavior of the real workload may rely on system inputs that may be impossible to reproduce exactly. Since the SW does not require real system inputs, its behavior can be reproduced. The user can make the SW accurately reproduce the workload characteristics of the actual workload by using the source code of the workload as a blueprint for constructing the SW. Using an SW when an actual application workload is available can have its disadvantages. First, since the SW must be constructed,

using it requires an additional cost, whereas the application software is already available and using it requires no additional cost. Next, tuning the SW to accurately represent the workload can be difficult and expensive. To achieve a high-level of accuracy, the user must first collect a large amount of data to characterize the workload and then use this data to tune the SW. Finally, the SW must be run on a dedicated target system. Therefore, an otherwise operational system must be taken off-line to be evaluated using an SW. If the actual application software is used as the workload, the system may be evaluated while it is in operation [27].

For the second type of evaluation, the SW is an ideal tool to use. Because the SW is parameterized, the user directly controls the workload characteristics whose effects on system performance are being studied. Thus, various combinations of workload characteristics may be produced by selecting the appropriate parameter values. The SW is particularly useful for experiments using full factorial or partial factorial designs because the different factors in the experiment can be controlled by specific parameters. The values of the parameters will then represent the various levels of the factors. The SW is superior to an actual workload for these types of evaluations. It is generally difficult to control specific characteristics of real workloads. To do so would require the user to identify exactly which code segments produced the workload characteristics of interest and then change the segments in such a manner as to produce the desired workload characteristics without adversely disturbing other workload characteristics. These problems are avoided by using an SW. The SW can be structured to localize the code which produces specific workload characteristics. This code can be written such that it has no side effects on other workload characteristics.

1.1 Research Objectives

In this dissertation, we study the design and implementation of SWs and tools for generating SWs for distributed real-time systems. Since, for some studies, the SW is to be representative of an actual application, we structure it like an actual application. To do this, we have developed a model of real-time workloads upon which the structure of the SW is based. The model describes the structure of the software which composes the workload. It describes the interactions of tasks, data structures, and interfaces to the environment. It also describes the internal structure of tasks with a degree of precision suitable to capture the timing behavior of each task. Since the SW is an executable version of the workload model, the accuracy of the model determines the representativeness of the SW.

For a distributed real-time system, an SW must be written for each processor. Writing and debugging a large number of SWs is a tedious and error-prone undertaking. Clearly, the repetitious and algorithmic nature of the process of generating SWs make it a candidate for automation. We have developed a *synthetic workload generator* (SWG) that automatically translates a textual representation of the workload model into an executable SW for a distributed real-time system. Our *synthetic workload specification language* (SWSL) contains a number of additional features including the ability to specify the SW in the context of an experiment. A single SWSL specification can describe a series of experiments where the parameters of the SW are different for each experiment.

We have designed and implemented an SWG that can compile SWSL specifications and produce SWs with the specified characteristics. We have developed a generic SW driver that controls the distributed SW as it executes on the target system. The driver is generic in the sense that its structure does not depend on the SWSL specification. We have identified and implemented the essential services that it must provide.

We also describe a set of experiments where we demonstrated the ability of the SW to act as both types of workload. It is capable of producing representative resource demands, and it has been used to produce specific, user-controlled resource demands for the experimental evaluation of a distributed real-time system. These demonstrations were performed on two different real-time computing systems, thus showing the generality of the SW design.

In summary, the objectives of this dissertation are to

- Develop a model of distributed real-time workloads,
- Develop a synthetic workload specification language,
- Implement a synthetic workload generator,
- Implement a driver to support distributed execution of the synthetic workload,
- Demonstrate the ability of the synthetic workload to be representative of an actual application, and
- Demonstrate the ability of the synthetic workload to produce specific, user-controlled resource demands for use in experimentation.

1.2 Outline of the Dissertation

This dissertation is organized as follows. In Chapter 2, we discuss the important properties of synthetic workloads and provide background information on the development of SWs. Chapter 3 describes the workload model that is used by SWSL. SWSL is defined in Chapter 4. The structure of the generic synthetic workload driver is discussed in Chapter 5. In Chapter 6 we describe a set of experiments that were run to demonstrate the ability of the synthetic workload generator to produce synthetic workloads that accurately represent real workloads. Chapter 7 presents experiments designed to demonstrate the ability of the synthetic workload generator to be used to generate synthetic workloads with specific workload characteristics for experimental performance evaluation of a distributed real-time system. Our conclusions are presented in Chapter 8.

CHAPTER 2

PRELIMINARIES

In this chapter we provide the context in which our research was performed. First, we discuss the important properties that an SW should possess. Our goal was to optimize these properties in our SW. Next, we present a background of SWs. We discuss the first SWs for data processing systems and trace subsequent SW development with special emphasis on SWs for distributed and real-time systems. We conclude the chapter with a summary of the background and discuss our approach to the development of SWs for distributed real-time systems.

2.1 Properties of synthetic workloads

To be useful for performance evaluation, an SW must possess a number of properties. The most important of these have been identified by Ferrari [24]. They are representativeness, flexibility, simplicity of construction, compactness, low usage costs, system independence, reproducibility, and compatibility. The following definitions are paraphrased from [24].

Representativeness is another term for modeling accuracy. It is the measure of how well the SW reflects the structure or behavior of the workload. *Flexibility* is the ability to alter the SW easily and inexpensively. *Simplicity of construction* refers to the cost and complexity of gathering the necessary information in order to design and construct the SW. It also refers the ease with which the user can specify and generate the SW. The *compactness* of the SW is a measure of the amount of system resources required to specify and use it. Compactness is generally proportional to cost and inversely proportional to representativeness. A compact SW is usually less expensive to use but less representative of the real workload. Representativeness requires that the SW contain more information about the workload that it is modeling. *System independence* is necessary if the SW is to be used to compare different systems or different versions of the same system. An SW should

be representative of the workload regardless of the system on which it is executing. As an aid to the performing of experiments, *reproducibility* is important. The degree to which the behavior of the SW can be reproduced from experiment to experiment is a good indication of the degree of control that the experimenter has over the SW. Finally, for the SW to be usable, it must exhibit *compatibility* between the SW and the system. This means that the SW must be made up of programs that can execute on the target system and effectively exercise the system's resources.

2.2 Background

A desire to implement these properties into a practical executable workload model has motivated the development of several synthetic workloads. The first SW was developed by Buchholz [10] for data processing systems. Buchholz's SW consisted of a single synthetic job. It was designed to model a commercial file update system common to business applications. Its inputs were a detail file and a master file. The SW would read a record from the detail file and find the corresponding record in the master file. Then, it executed a compute loop to simulate processing of the records. Finally, it wrote the updated master record. The synthetic job was parameterized to allow it to model any of a number of jobs with similar structure but different workload characteristics. The parameters included the number of records in the detail and master files and the number of repetitions of the compute loop. The job could be made to execute a number of times in succession with varying values for the parameters.

Buchholz's idea was expanded by Wood and Forman [70], who created a synthetic *job stream* composed of a collection of Buchholz's synthetic jobs executing concurrently. This job stream was an improvement over the sequential execution of jobs in that the interactions between a number of jobs with differing workload characteristics could be modeled. They also added parameters to the synthetic job to allow the user to specify how many data files to be written when the compute loop had completed and to allow a number of lines to be printed for each record updated. Sreenivasan and Kleinman [62] further refined the synthetic job by adding parameters to specify the block size of the I/O buffers and the size of the records. They then used multiple copies of the same job, each with different parameter values, to create the desired workload. They also described a technique whereby the characteristics of an actual workload may be used to determine the values of these parameters. This technique was later adapted by Haring *et al.* [32], for the evaluation of a mainframe computer.

Schwetman and Brown [57] duplicated Wood and Forman's system but added a synthetic job generator. This generator was a program that submitted synthetic jobs to the system in a manner designed to simulate the arrival patterns of jobs to the system in a real workload. It also allowed the value of the workload parameters to be altered for each job. In order to increase the representativeness of the workloads being generated, Lucas [45] proposed that a library of synthetic job modules be compiled. This library would contain modules to represent the different types of jobs that are present in the workloads of general-purpose systems. He gave examples of the types of modules that would be contained in the library. The modules were chosen to measure different aspects of system operation, including compiler attributes, operating system attributes, and program execution attributes.

This scheme was generalized in the APET system, which provided a language for defining an SW [4]. APET modeled the workload as consisting of a set of *phases*. Each phase was essentially a synthetic task. Phases ran concurrently and interacted with each other. A phase was composed of a number of *steps*. Each step was a program function that executed a specific workload operation. Each phase repeated in a cyclic pattern for a user-specified period of time, and during each cycle the steps of a phase executed sequentially. Parameters of the model allowed the user to select which phases and steps were to execute. The operations supported by APET were restricted to CPU usage and file manipulation.

Walters [64] developed a system called SKET in which representative benchmark kernels were constructed based on program flowcharts. Kernels for individual program functions were chosen by the user and then compiled to produce performance formulae that were used to derive performance data. A compiler to produce an executable benchmark was proposed but not implemented.

Dujmovic [19] introduced the idea of *monoresource synthetic programs*. Each program in the SW exercised a single resource continuously for the duration of its execution. These programs were combined in proportions expected to accurately model the resource usage of the modeled workload. The performance of the system was modeled as a linear combination of the workload characteristics.

A high-level language description of an SW was developed by Singh and Segall [59, 60] for the Pegasus performance evaluation system. This language, called the B-language, represented the workload as a UCLA graph, a form of dataflow graph. The user was able to specify the structure and behavior of the workload at a high level. The language defined tasks and their interactions. The internal structure of each task was specified as sequences of operations and control constructs, such as branches and loops. It was intended that a specification in the B-language be compiled by a synthetic workload generator (SWG) to

create the executable SW. The SWG for Pegasus was never implemented.

Advancements in modeling individual workload characteristics have also occurred. For example, Babaoğlu [3, 2] and Ferrari [25] developed techniques to allow an SW to produce representative memory referencing patterns for virtual memory systems. Ferrari has made a number of contributions in the development of SWs [23, 24, 26, 27, 28], including the definition of representativeness that we use in Chapter 6 and the development of models of workloads of interactive systems.

Calzarossa, Italiani, and Serazzi [11] focused on the static (structural) and dynamic (execution) representativeness of the SW. They characterized the structure of a workload according to the resource usage parameters. Then, they used clustering techniques to find representative jobs from the workload. They modeled these jobs and reproduced their execution behavior stochastically.

A number of SWs have been developed to act as terminal emulators [29, 8, 47]. These SWs generally emulated behaviors of clients in a client-server environment. They were driven either stochastically or by traces.

All of the above SWs were developed for general-purpose computing systems. SWs for real-time systems are scarce. One example was the SW for NASA's Fault-Tolerant Multiprocessor (FTMP) [21, 22]. FTMP's SW was designed to exercise the system and perform a limited number of timing measurements. It defined the workload as a number of periodic tasks divided into three *rate groups*. A rate group was a collection of periodic tasks with the same period which were invoked at the same time. The periods of the rate groups were aligned at *major cycle* bounds. A major cycle was the least common multiple of the lengths of the periods. Thus, at the beginning of each major cycle, all tasks were invoked simultaneously. The deadline for each task was equal to the length of its period. The usefulness of this SW was restricted by the fixed synthetic program structure, fixed deadline policy, lack of aperiodic tasks, and few locations where timing data was collected. This SW was ported to HARTS at the University of Michigan by Woodbury [71].

Support for SWs appeared in Scheduler 1-2-3, a schedulability analyzer developed at Carnegie Mellon University [63]. Scheduler 1-2-3 was capable of producing workload tables as a part of its schedulability analysis. The main workload parameters in the table were the period, priority, and phase (alignment of periods) of the tasks. These tables could be included into the SW that was used to evaluate the ART Real-Time Testbed. A similar SW was used by Wendorf [67] to analyze the performance of real-time scheduling processor running the Mach kernel.

Recently, two benchmarks for real-time systems have emerged. Benchmarks are

SWs with no adjustable parameters and, thus, fixed behavior. The first is Rhealstone, a proposed benchmark for real-time systems [40]. Rhealstone measures the performance of six important functions of a real-time system: task switching time, preemption time, interrupt latency time, semaphore shuffling time, deadlock breaking time, and datagram throughput time. It was suggested in [40] that the performance of the system under a real workload may be estimated by a linear combination of the measured values.

The second real-time benchmark is the Hartstone Distributed Benchmark [37]. Hartstone was created specifically to benchmark the communication performance of real-time distributed systems. It provides quantitative measures of communication performance. Its execution is composed of a series of experiments aimed at measuring system capacity. The experiments successively increase the load on the system until it fails to provide the required service.

Both of these benchmarks are useful for measuring certain aspects of real-time system performance. However, as is the case with most benchmarks, the set of measured performance indices is limited. There is no flexibility to measure other indices that may be important to a given study. A fixed set of performance indices is insufficient for determining the total performance of the system under a real workload.

2.3 Summary

Much work in SWs for non-real-time systems has been focused on data processing systems. Most of the SWs were trivial extensions to Buchholz's SW until Ferrari made several significant contributions to the theory of SWs. He developed techniques for constructing SWs to model specific workloads and for evaluating the representativeness of the SW. The most important developments in SW construction techniques were by Lucas, Walters, and Dujmovic. Lucas and Dujmovic refined the concept of synthetic operations stored in libraries. Walters made the first advances in achieving representativeness by using the program's structure as a template for the SW's structure. Singh and Segall applied a number of these techniques in creating the B-language for a distributed system. Unfortunately, they never implemented an SWG to compile the B-language. All reported examples were hand translated before execution. Thus, they were not able to show that their approach was feasible in practice. They showed no correlation between UCLA graphs and actual software structure, and no attempt was made to demonstrate representativeness.

Little progress has been made by others to apply these principles to the creation of SWs for distributed real-time systems. The real-time SWs discussed above provide little

more than the ability to create periodic synthetic tasks. Singh and Segall [59] claimed that their SW could model real-time workloads. They presented an example that was labeled as a real-time application. It consisted of a feedback loop for processing sensor data. While this application resembled some components of real-time workloads, there were no timing constraints specified, and no time-critical behavior was analyzed. At no point was it demonstrated that the B-language contained explicit support for modeling real-time applications or that the workload was representative of any real-time workload.

In this dissertation, we build upon this past work to create a tool for generating SWs for distributed real-time systems. The characteristics and requirements of distributed real-time systems require innovative applications of known features and the development of new features in the design of SWs. For example, like Walters, we use the structure of the software as a template for our SW, and, like Lucas and Dujmovic, we make use of a library of synthetic operations. However, our SW must display the characteristics of real-time software, and the library must contain operations common to real-time workloads. We combine these features to produce a high-level language that can be used to specify SWs easily and compactly. We pay particular attention to representativeness and, unlike the developers of previous SWs, the requirements of experimental evaluation. The language and the SWs generated are designed to make experimentation easier, and special attention is paid to improving the statistical properties of the SW.

CHAPTER 3

MODELING REAL-TIME WORKLOADS

3.1 Introduction

Our workload model is intended to describe real-time workloads in sufficient detail to be used as the basis for generating SWs. To be an accurate representation of the workload, the model must capture all structural and behavioral details of the workload. The structure and behavior of the workload directly affect the values of the performance indices that are measured during experiments. Changes in the workload cause changes in the values of the performance indices. It is by characterizing these changes that one evaluates the system. The workload model provides a formalism that allows the user to express the connections between the workload, its characteristics, and the measured performance indices.

A real-time system is a computing system where the value of a computation depends not only on the logical correctness of the results, but also on the time at which the results are produced. This definition describes a class of systems with characteristics that set them and their workloads apart from general-purpose systems [37, 50, 42, 15, 13]. They are usually embedded in a larger system that performs a particular function. The real-time computing system serves as the controlling computer for this larger system. The real-time system is designed to execute specific application software required to control the larger system. All tasks are predefined and their parameters are usually known *a priori*. The control activity consists of accepting frequent or continuously arriving inputs from sensors and, in response, producing output to actuators and/or display devices. These responses must occur soon enough after the input to meet the physical constraints of the system. The system must also accept inputs at random times due to operator commands and exceptional conditions. The hardware of the system may be distributed, consisting of a number of processors each connected to a variety of I/O devices. Distributed systems exhibit great potential for high performance and high reliability, two properties that are essential for real-time systems.

To provide the required services, the real-time workload consists of a number of *periodic* tasks which handle the periodic I/O associated with process control. There are also *sporadic* tasks which execute in response to the aperiodic events. The requirements of the system are such that the responses to inputs must occur within predetermined time intervals, i.e., responses have deadlines. There may be a number of distinct states in which the system operates. Tasks may behave differently depending on the state. Although some of the tasks may execute independently, they will often be required to communicate with one another and exchange data.

Previous approaches to modeling workloads consisted of capturing the behavior of the workload using queueing networks or describing the workload in terms of a vector quantifying the workload characteristics [24]. However, the properties of a real-time workload are not accurately modeled by a queueing network or as a simple vector of workload characteristic values because these techniques model average case performance. Therefore, they cannot capture the details of the timing characteristics of the workload. To model a real-time workload, we must accurately describe the details of the workload that specifically influence the time-related aspects of the system. The model should express the tasks' timing, resource usage, and interaction characteristics. The timing characteristics include task execution times, deadlines, and scheduling parameters. The resource usage characteristics should include access priorities, preemption policies, and the quantity of the resources used along with the timing characteristics (e.g., pattern and duration) of that usage. Task interactions include both direct communication and resource sharing. Since standard queueing models and simple vectors of workload parameters are neither powerful nor expressive enough to model real-time workloads, a different, more expressive model is needed.

This chapter describes a workload model with sufficient expressive power to describe real-time workloads. In Section 3.2, we discuss some issues that influence the model design. We then describe the details of our model in Section 3.3 and conclude with Section 3.4.

3.2 Model Design Issues

We have constructed our model to accurately capture the structure and behavior of a real-time workload. The workload is described in terms of a dataflow graph, a notation commonly used to specify software for distributed real-time systems. The model is a generalization of the rapid prototyping language PSDL developed by Luqi, Berzins, and Yeh [48] and three structured analysis (SA) notations: ESML [9], Ward and Mellor's transformation

schema [65], and the Boeing/Hatley notation [33]. These SA notations are commonly used in CASE tools to specify and analyze the requirements and structure of real-time software. The dataflow model captures the basic aspects of the workload, e.g., parallelism of tasks and interactions between tasks, and allows for modeling at multiple levels of abstraction. These features provide a generality which makes our model flexible and thus more widely applicable. Thus, it is capable of modeling the features of a number of SA, rapid prototyping, and other notations, e.g., [52, 30, 49], and can be used to describe a wide range of real-time workloads which have been specified using these notations. Our model extends these notations to specify the timing and resource usage properties of the workload.

The model was based on SA and rapid prototyping notations for the following reasons:

- At the time a prototype system is ready for evaluation, it is likely that the system designers will only have a high level specification of the proposed application software, i.e., the SA model. This model will generally be a good approximation of the structure of the workload [33]. Thus, by using a similar model for our SW, we can produce an SW which will closely approximate the structure and behavior of the proposed software. The experimental evaluations performed using this SW will then provide useful and meaningful results. Similarly, developers of experimental systems can make use of published workload specifications, e.g., [46, 44, 51, 69], to produce representative SWs to be used to evaluate their systems.

- Since the workload is modeled at a high level of abstraction, the model is system independent. The model does not contain any information that is particular to a given hardware architecture or operating system. Therefore, a workload specified using our model is portable and may be used to comparatively evaluate different systems.

- As real-time software becomes more complex, the use of structured methods to design the software will become widespread. The design process will be supported by computer-aided software engineering (CASE) tools [20, 53]. Our approach allows the SWG to become an integral part of a CASE tool. A number of CASE tools use SA and similar notations. Hence, high-level software designs created by CASE tools can be translated to our model and used by the SWG to create SWs. The SWs thus produced will be akin to a rapid prototype. The difference is that, while the rapid prototype is aimed at demonstrating the functionality of the software from the user's viewpoint, the SW is aimed at demonstrating the resource utilization behavior of the software from the system's viewpoint.

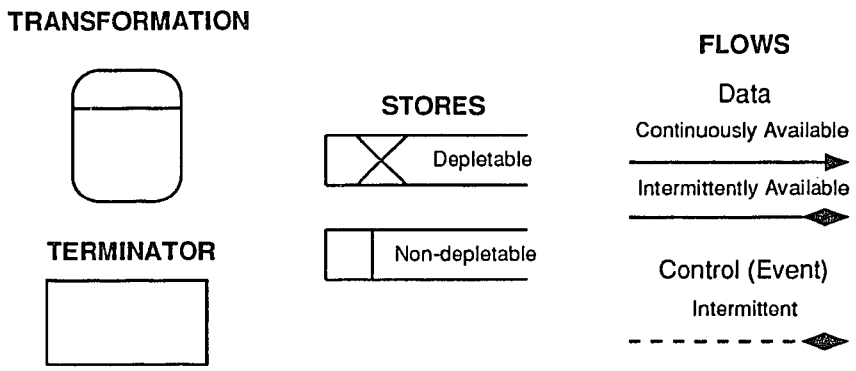


Figure 3.1: Model components

- Using an common notation allows us to make use of existing off-line analysis and simulation tools. A number of tools have been developed to execute and analyze SA specifications directly [7, 56, 66] or indirectly by transforming the SA model into a Petri net [54, 55]. The specification of the workload using our model can be transformed into an SA specification or equivalent Petri net and then be used to derive performance measurements using either simulation or analytic techniques.

3.3 The Workload Model

A real-time workload is defined as a 5-tuple, (T, S, R, F, D), where T is a set of *transformations*, S is a set of *stores*, R is a set of *terminators*, F is a set of flows, and D is data. These workload objects will be described in detail in the following sections. The graphical representation of all components are shown in Figure 3.1. These symbols are taken directly from the graphical representation for ESML.

3.3.1 Transformations

The set of transformations T represents the work done by the workload. Transformations encapsulate both the processing of data and the control logic of the workload. We define $T = \{t \mid t = (I, O, p, \phi, \pi_1, \ldots, \pi_N)\}$ where I is the set of inputs, O is a set of outputs, p is a processor identifier, ϕ is a function, and $\pi_1, \ldots, \pi_N$ are N system-specific parameters where N is an integer whose value depends on the target system. The transformation receives data or control signals on its inputs, I, and produces data and/or control signals on its outputs, O.

The behavior of the transformation is determined by the function ϕ. The transfor-

mation may represent any function for data processing and/or any control structure. The combination of data flow and control in a single transformation is a generalization of the SA and rapid prototyping notations. This control mechanism is more powerful than the mechanisms defined for ESML and Singh and Segall's B-language. It is capable of modeling control constructs such as state machines, control flows, and control transformations. Thus, various mechanisms for specifying system state and state-dependent operations may be modeled.

The p in the definition of t represents the assignment of a transformation to a specific processor. All transformations are considered unique. Therefore, replicated transformations in fault-tolerant systems are modeled individually.

The timing and selection of inputs and outputs are determined by the internal structure and behavior of the transformation. Transformation behaviors are not restricted to the model of "trigger, compute, produce output" which is common to dataflow specifications. Instead, transformations are free to perform inputs and outputs at any time during their execution. Based on their internal logic, they are also able to select whether or not to read a given input or produce a given output. This flexibility in defining task interactions is necessary when specifying SWs for real-time systems. If the SW is to be representative of real applications, the synthetic tasks must accurately reproduce the complex timing and resource sharing dependencies between tasks. This accuracy can not be obtained from a simple dataflow model [68]. It requires the more detailed specifications allowed by this model.

The function specified by ϕ is defined based on the D-structures described by Ledgard and Marcotty [41]. The set of D-structures is a small functionally-complete set of control constructs for programs. They consist of simple operations, composition of D-structures, a conditional control construct, and a loop construct. A simple operation is any computation, system call, or input or output statement, etc. These are the smallest units of execution in the model. Composition is the simple sequential execution of two D-structures. For two D-structures `s1` and `s2`, composition is represented as `s1; s2`. The conditional control construct is the "**if** `condition` **then** `s1` **else** `s2`" construct. The looping construct is the "**while** `condition` **do** `s`" construct. With these constructs, all other control constructs may be realized [41].

The complete specification of a transformation depends on the system upon which it is to be implemented. Different operating systems require different information to create and schedule the implementation of the transformation. Therefore, the transformation specification includes a number of system-specific parameters, π_i. These parameters may

include scheduling parameters, resource requirements, functions for exception handling, etc. The number of π_i parameters, and thus the value of N, depends on the target system. The model defines as many π_i parameters as are needed to specify the implementation of transformations on a given target system.

3.3.2 Stores

Stores model all objects which can contain data. These objects include data structures, files, sockets, pipes, etc. A transformation passes data to another transformation by placing the data in a *store* from which the other transformation reads the data. Formally, we define $S = \{s \mid s = (I, O, p, \pi_1, \ldots, \pi_N)\}$ where I is the set of inputs, O is a set of outputs, p is a set of processors, and $\pi_1, \ldots, \pi_N$ is a set of N system-specific parameters where N depends on the target system. The I, O, and p values have the same meaning as in the definition of T. The π_i values define parameters required to specify the implementation of a data structure modeled by the store. They define its storage properties: e.g., element size, storage policy, and access policy. S may be divided into two disjoint subsets such that $S = S_d \cup S_n$ where S_d is the set of *depletable* stores and S_n is the set of *nondepletable* stores. Depletable stores represent objects such as stacks and queues where a data element is removed from the store when it is read. A nondepletable store represents an object like shared memory which retains the data value after a read. The reader receives a copy of the data.

3.3.3 Terminators

Terminators serve as the interfaces between the workload and the environment. We define $R = R_i \cup R_o$, where R_i is the set of input terminators and R_o is the set of output terminators. $R_i = \{r \mid r = (O, p, \pi_1, \ldots, \pi_N)\}$ and $R_o = \{r \mid r = (I, p, \pi_1, \ldots, \pi_N)\}$ where I is the set of inputs, O is a set of outputs, p is a set of processors, and $\pi_1, \ldots, \pi_N$ is a set of N system-specific parameters. The I, O, and p values have the same meaning as in the definition of T. The π_i values define parameters required to specify the characteristics of the terminator. They specify the interface between the workload and the environment. They define the type of the interface, the size of the data elements that it handles, and the minimum sampling interval or minimum data acceptance interval. The R_i and R_o terminators are referred to as *sources* and *sinks*, respectively. A source terminator represents a point where data is received by the workload from an external object. Typical examples of such objects are sensors and operator controls. Sink terminators represent locations where data or control signals are sent to an external object by the workload. Actuators and

displays are examples of external objects which may be represented by sink terminators. Terminators may also be paired to represent resources such as external files or databases which have both inputs and outputs.

3.3.4 Flows

Flows are used to connect objects. Thus, we define $F = \{(s,d) \mid s,d \in T \cup S \cup R\}$, where s is the source of the flow and d is the flow's destination. Flows are the paths used to transfer data and control signals from one object to another. We define three types of flows in SWSL: $F = F_c \cup F_i \cup F_e$, where F_c and F_i are two sets of value-bearing flows and F_e is the set of non-value-bearing flows. The value-bearing flows are *data flows*. They are distinguished according to whether the data values are continuously available (F_c) or intermittently available (F_i), i.e., available only at discrete instances of time. These value-bearing flows will be referred to as *continuous* data flows and *intermittent* (or *discrete*) data flows, respectively. The non-value-bearing flows (F_e) are *event* flows, and they carry intermittently available *signals*.

3.3.5 Data

Data is defined as the unit of information in the system. We define $D = \{d \mid d = (v,s)\}$. Each unit of data has a value, v, and a size, s.

3.3.6 Interconnection Rules

Construction of a workload using our model is based on the construction of one using ESML [9]. The model construction rules are specified formally by the definitions of the flow types:

$$F_c \subseteq (T \times S_n) \cup (S_n \times T) \cup (T \times R_o) \cup (R_i \times T).$$

Continuous data flows may be used in either direction between transformations and nondepletable stores, and between transformations and terminators:

$$F_i \subseteq (T \times S) \cup (S_d \times T) \cup (T \times R_o) \cup (R_i \times T).$$

Intermittent data flows may be used to connect transformations to any type of store, and are used to connect depletable stores to transformations. They may also be used to connect transformations and terminators in either direction:

$$F_e \subseteq (T \times T) \cup (T \times R_o) \cup (R_i \times T).$$

Event flows may be used to connect transformations with each other or to connect transformations with terminators in either direction. There is only one additional rule which cannot be defined using the notation above. It states that a transformation must have at least one input and one output flow.

The above rules have some implicit consequences. First, a flow must have a transformation at one or both ends. Transformations are the sole active components of the workload. It is through the actions of transformations that data is moved through the workload. Second, data flows may not be used to connect transformations directly. To pass data from one transformation to another, the data must first be written to a store. The receiving transformation then reads the data via a data flow. This data passing model is accurate since, in real systems, any data passed between two tasks must be buffered somewhere. This buffer may be in local, global, or system memory, but it always exists. Therefore, we require that it be modeled. Third, signals have no value associated with them. Therefore, they may not be kept in stores. Hence, event flows may not be inputs or outputs of stores.

3.4 Summary and Conclusions

In this chapter, we have described our model of real-time workloads. The model is general enough to describe a wide range of workload structures which might be used in real-time systems without being overly restrictive. While the model is sufficiently expressive to be used for many types of performance evaluations, it is designed particularly for the specification of real-time SWs. For simplicity, ease of use, and added representativeness, this model is based on SA notations currently being used for real-time software development. By using an SA-based model, we simplify the modeling problem of translating high level software specifications into SWSL. The users do not have to translate their software specifications into a completely foreign, and possibly incompatible, model to make use of the SW. Some translation is necessary because the model has been made more general to avoid being tied to a given SA notation. However, the generality of the model improves portability and makes the model compatible with a wider range of high-level system specification models. Since the model is based on actual software specifications, it accurately models the structure and behavior of the actual workload.

CHAPTER 4

SYNTHETIC WORKLOAD SPECIFICATION LANGUAGE

4.1 Introduction

In this chapter, we present the Synthetic Workload Specification Language, SWSL. SWSL is a language designed specifically to specify SWs. Its design was influenced by four requirements. First, it should possess the desirable properties of SWs. These properties, as listed in Section 2.1, are representativeness, flexibility, simplicity of construction, compactness, low usage cost, system independence, reproducibility, and compatibility. As we describe the various features of SWSL, we will discuss how each improves one or more of these properties.

Second, SWs are to be used for the experimental evaluation of distributed real-time systems. Therefore, SWSL should support the process of experimental evaluation. It should allow the user to define experiments and should provide mechanisms to define SWs with useful statistical properties.

Third, the SWs will execute on an embedded real-time systems. Due to the timing requirements of the system, the SW must be able to execute without interactive interference from the user. Therefore, the language must specify any workload characteristics that should vary at run-time. These characteristics are compiled into the SW.

Fourth, an SWSL specification is compiled by a synthetic workload generator (SWG). Therefore, as with any computer language, the syntax and semantics are designed so they may be easily compiled and errors in the input files detected and located. The SWG will be discuss in Section 4.4.

This chapter is organized as follows. In the next section, we discuss the important issues in specifying SWs which are addressed by SWSL. In Section 4.3, we define SWSL. Section 4.4 describes how SWSL is used by our SWG to produce SWs. We conclude with Section 4.5.

4.2 Specification of Synthetic Workloads

Before discussing the details of SWSL, we first present the underlying concepts of its design. These concepts are driven by the requirements outlined in the previous section.

Abstraction

SWSL takes great advantage of the primary property of SWs: abstraction. SWs are useful in experimental evaluation because they abstract out details of a workload and produce only those resource demands which are required for a given evaluation. For example, to evaluate the scheduling policy of a real-time operating system, each task in the SW might abstract out the specific computations performed by a task in an actual workload and simply reproduce the total amount of CPU time required for computation. SWSL uses abstraction to achieve compactness and much of the simplicity of the SW specification.

If a task in the SW is not performing the actual computations of the workload, it can not produce the correct results of the computation for use by other tasks. Those tasks also abstract computation, so the value of the data is irrelevent. Therefore, the SW also abstracts data. In the SW, only the size of the data is important, because we only consider the resources required to store the data. The effect of data on the behavior of the workload is modeled stochastically. An advantage of this abstraction is that the SW can operate without requiring actual input data and tasks can produce the resource demands due to computation without executing the exact algorithms from the modeled workload. A disadvantage is that low-level, data-dependent behaviors of the workload are more difficult to model using the SW. We provide mechanisms in SWSL to allow the user to model these behaviors, but these mechanisms require greater programming effort by the user and more information about the workload being modeled.

Representativeness

SWs specified by SWSL are based on the workload model described in Chapter 3. By basing the SW on a workload model we improve the representativeness of the SW. To measure representativeness, we use a performance-based metric. By this metric, an SW is representative of a workload if the performance of the system (as measured by a set of performance indices) while executing the SW is the same as the performance while executing the workload [27]. However, "[e]xcept for certain cases . . ., this definition of [representativeness] does not directly suggest a method for designing an artificial workload" [27]. Given this observation, we use a structure-based method for constructing a representative SW.

That is, the SW is specified and constructed such that its structure models that of the workload. Other researchers [64, 4] have successfully produced sufficiently accurate and flexible benchmark programs for uniprocessor systems by modeling the structure of the actual workload. We expect the technique to be successful for distributed systems. The structure-based representation is complemented by selecting the appropriate π_i parameters for each object and assigning appropriate values to the parameters. These parameters determine the characteristics of the object as it is presented to the system. Parameters specify the resource requirements and the time-dependent behavior of the objects. By providing the SW with the same structure as the workload being modeled and by tuning the parameters which determine the behavior, we are able to produce a representative SW. The level of representativeness may be measured by the performance-based metric. The ability of SWSL to produce representative SWs is demonstrated in Chapter 6.

Flexibility

Flexibility is another important characteristic of SWSL. If SWSL is to be useful for experimentation, it must be flexible. The user must be able to easily change the values of specific workload characteristics. This ability requires that SWSL be able to produce SWs with a wide range of resource requirements and behaviors. Flexibility within a narrow range of behaviors is of limited benefit. Flexibility is provided primarily through the parameterization of the objects in the workload. All significant workload characteristics may be controlled by changing the values of the proper parameters. In many cases, the user can make significant changes to both the structure and the behavior of the workload by changing a few parameter values. More importantly, the user can produce small, incremental changes to specific workload characteristics with little effort. Many evaluations involve measuring the performance of the system for various values of a given workload characteristic. Changing the value of a workload characteristic is generally as easy as changing the value of one parameter.

SWSL does not restrict which behaviors and structures can be included in the workload. SWSL was developed with a specific set of π_i parameters needed to specify SWs for the target systems available to us. The user can add or delete π_i parameters in the specification of workload objects if those parameters are needed to specify the implementations of workload objects on their target system. In addition, the user can specify exact C language code within the function for a given synthetic task. This feature would be used to produce behaviors at a lower level than can be specified by SWSL.

Flexibility is also improved by taking advantage of the definition of transforma-

tions in the workload model. For each transformation, the inputs, outputs, and function are defined separately. In SWSL, the definitions of functions are decoupled from the definitions of transformations. Therefore, the behavior of a transformation may be altered very simply by specifying a different function for it to execute. The only requirement is that the function operate on the same number and types of inputs and outputs as are defined for the transformation. Furthermore, the functions executed by different transformations need not be unique. A single function definition may suffice for a large number of transformations, thus resulting in a more compact and easily constructed SW specification.

Object Templates

The workload model defines each object uniquely. SWSL makes the specification of objects more compact by providing a simple mechanism whereby one can produce many instances of an object from a single *object template.* All instances of the object will have the same values of all the π_i parameters. There are two uses for object templates. The first is to specify an object which represents a member of a class of objects with similar parameters. This technique is a common one in workload characterization [58] and has been used often to specify SWs, e.g., [1]. A set of n parameters are selected to define the important characteristics of the workload tasks. For each task, these parameters are measured and the task is plotted in the n-dimensional space defined by the parameter vector. A clustering analysis is performed to partition the tasks into groups with sufficiently similar characteristics. Then, a small number of tasks from each group are selected to represent that group. These representative tasks are used as templates. The number of instances of the task that are produced is proportional to the size of the cluster being represented relative to the size of the entire workload. This technique reduces the number of task specifications that must be written and thus makes the workload specification more compact. The second reason for using templates is to represent objects which are replicated for purposes of fault-tolerance. In a fault-tolerant real-time system, multiple copies of an object will execute on separate processors. They perform the same calculations and the results are combined via voting. Using this technique, the system can mask a given number of faulty processors.

Support for Experimentation

Experimental design is supported through several SWSL design features. First, we differentiate between the SW and the measurement mechanisms used to collect data in the evaluation. The only function of the SW is to serve as the workload for the target system; it does not provide any mechanisms for measuring performance. In this way, the

SW is different from a benchmark program, which not only exercises the system but also measures the performance of the system while it is being exercised. The SW is designed to work harmoniously with performance evaluation mechanisms. Therefore, the user is free to choose any appropriate measurement mechanism. If a software monitor is being used which must be executed as a user task, the monitor can be specified as the function for a transformation. The monitor will be compiled into the SW and function normally on the target system.

Second, the typical experiment using the SW consists of a number of runs, each of which is composed of the following steps: the SW code is generated and compiled from the specification; the executable code is downloaded to the target computer; the SW is executed; measurements are made and data is collected. Most such experiments will be aimed at measuring the performance of the system as a specific workload parameter (or set of parameters) is varied. Under the above scenario, each run of the experiment would involve repeating the set of steps listed above for each new value of the parameter(s). To reduce the time required to perform such experiments, SWSL supports a multiple-run facility. For each parameter of the SW, the user may specify a list of values. When the SW is first invoked, the first value provided for each parameter is used. Once the run is completed, the SW pauses to allow time for measurement mechanisms to be reset and initialized for the next run. To begin the next run, it reinitializes and executes again. This time, it uses the next value in the list for each parameter. The reinitialization between runs is necessary to insure statistical independence of values measured in consecutive runs. The only state preserved between runs is the run count. This facility reduces the time-consuming compilation and downloading processes to a single compilation and download for a series of runs.

Specifying the parameters for all runs at compile time is preferred for use in real-time systems. Changing parameters at run-time requires the presence of an agent which can interpret user commands and make the appropriate changes to the system and application data structures on the target machine. The interference caused by the agent would adversely affect the timing characteristics of the executing SW. In addition, the agent would have to be custom developed for each different target architecture, as it would have to make use of the communication and system functions which are peculiar to each. This added effort reduces the ease with which the SW can be ported to and used on a new system.

SWSL has additional mechanisms to support experimentation that have been absent in previous SWs: reproducible experiments, independence of events within a given experiment, and statistically independent experiments [24, 34]. As described above, the SW reinitializes the experiment between runs. In addition, it simulates data-dependent ac-

tivities stochastically. Each such activity makes use of a separate random number generator stream. This technique allows independent objects to exhibit reproducible, independent behavior. This feature is especially important when evaluating multiprocessor systems where nondeterministic behavior is common. Sharing a random number stream would cause correlation between actions that would be irreproducible in a nondeterministic environment.

4.3 SW specification files

The SW specification is divided among three different input files. The three files specify the task graph, the experimental parameters, and the task functions, respectively. Although each file has its own particular syntax, there are some constructs that are common to all the files. The SWSL files are divided into sections. The two common sections are the EXTERNS[1] and the CONSTANTS. The graph and function files have an EXTERNS section, and all files have a CONSTANTS section. The EXTERNS section is used to declare functions and objects that are defined outside that file. These include task functions (declared as type FUNC), synthetic operations (OPER), random number generators for specific distributions (DIST), and the identifiers for the processors on which the objects are to be located (PROC). Task functions are defined in the functions file and transformations are defined in the graph file. Therefore, functions are considered external to the graph file and are required to be declared in the graph file's EXTERNS section. Both the synthetic operations and the distributions come from libraries which are linked with the SW during compilation. The processor identifiers are used by the SWG to designate which executable files are to be downloaded to which processors. The CONSTANTS section contains definitions of constants. The use of constants is common in programming languages, e.g., C. It is especially important when specifying SWs since it allows the user to localize changes to parameter values.

The programs that make up the SW are synthetic. They do not perform useful calculations; they do not process real data. Hence, data is redefined to have a single parameter, size. Data values are ignored. The effects of data values on the behavior of a real workload are simulated stochastically. SWSL uses a library of random number generators for different distributions. A call to a distribution function, with appropriate parameters, may be used as the value to many workload parameters. It may also be used as the loop count for the looping construct in the task function specifications (See Section 4.3.3). At run-time, the value will take on the value calculated dynamically by the distribution function.

In addition to these features, SWSL contains some basic language constructs that

[1] In this dissertation, all SWSL keywords will be given in uppercase.

enhance its ability to specify complex workloads. Comments and an "include" facility are supported to allow users to document and organize files according to their own style. SWSL supports simple arithmetic expressions where operands are scalars, constants, or distributions. Expressions may be used anywhere a numeric value is required, e.g., as values for parameters.

4.3.1 Graph file

The graph file describes the task level structure of the SW in terms of the workload model. It defines the transformations, stores, and terminators, along with their parameters. The graph file is divided into `EXTERNS`, `CONSTANTS`, `OBJECTS`, and `DEFINITIONS` sections. The various workload objects are declared in the `OBJECTS` section and defined in the `DEFINITIONS` section. An object is defined by listing its parameters with their values in the following format.

<object name> [
<parameter list>
];

The <parameter list> is the series of definitions of parameter values. This list defines the π_i, ϕ, I, O, and p parameters for the objects. These parameters are discussed in the following sections.

The π_i Parameters

Each π_i parameter is specified using the following format.

<parameter keyword> = $<\text{value}_1>$, $<\text{value}_2>$, ... , $<\text{value}_n>$;

Where `<parameter keyword>` is the parameter name, and $<\text{value}_i>$ is the value that the parameter is to take on for the i-th run. If fewer values are listed than the number of runs, then the last value in the list will be used for its corresponding run and all subsequent runs. One important use of this feature is for compactly specifying parameter values that remain constant across runs. For example, if the period of a task is always 10 milliseconds and there are seven runs in the experiment, then the `PERIOD` parameter of the corresponding transformation may be defined as

PERIOD = 10;

instead of

PERIOD = 10, 10, 10, 10, 10, 10, 10;

Parameter Name	Description
START_TIME	Time at which the transformation is first invoked.
PERIOD	The period of a periodic transformation.
SPORADIC	The function that returns the time between invocations for a sporadic transformation.
DEADLINE	The deadline of the transformation, either a scalar value or distribution name.
PRIORITY	The priority of the transformation. Required when the target system uses priority-based task scheduling.
ACTIVE	Indicates whether the transformation executes during each run.
NAME	String to be used to identify the transformation to the system.

Table 4.1: Transformation parameters.

Parameter Name	Description
TYPE	The type of the store: `DEPLETABLE` or `NONDEPLETABLE`.
ELEMENT_SIZE	The size (in bytes) of each element in the store.
CAPACITY	The storage capacity of the store measured in number of elements.
ACCESS	The access policy for the store.
POLICY	The storage policy for a depletable store, e.g., `FIFO`, `LIFO`, or `PRIORITY`.
NAME	String to be used to identify the transformation to the system.

Table 4.2: Store parameters.

All time values in SWSL are measured in milliseconds. Therefore, no indication of time unit is necessary in the specification.

The π_i parameters for transformations, stores, and terminators are shown in Tables 4.1, 4.2, 4.3, respectively. Transformation parameters indicate the transformation's scheduling requirements. The store parameters specify the type of data in the store and the access methods to be used. Stores represent all information repositories and data channels. Thus, the parameters have been selected such that they are orthogonal, and combinations of values may be used to represent different storage objects. Similarly, for terminators, we have chosen parameters to allow a range of terminator types.

The selection of π_i parameters is not fixed. We selected these parameters to specify the system-dependent characteristics of real-time workloads on HARTS[2]. HARTS is

[2]HARTS is the target system for which the SW was first developed.

Parameter Name	Description
TYPE	The type of the terminator: SOURCE or SINK.
ELEMENT_SIZE	The size (in bytes) of each element generated or accepted by the terminator. May be a constant or variable value.
START_TIME	Time at which the terminator becomes operational.
RATE	The time between data arrivals at a source terminator or the minimum time between data acceptances for a sink terminator. Either a constant value or a value taken from a specified distribution.
ACCESS	The access policy for the terminator.
NAME	String to be used to identify the transformation to the system.

Table 4.3: Terminator parameters.

discussed in Chapter 7. If required, parameters may be added to the language by updating the list of recognized parameters in the SWG source code. However, the selection of parameters will generally be fixed for a given installation of the SWG. New parameters are not added dynamically simply by adding them to the parameter list in the SWSL specification.

The ϕ Function

The ϕ functions for transformations are declared using the FUNCTION parameter keyword followed by the list of names of the functions to be executed in each run. These functions are defined in the functions file.

The I, O, and p Parameters

Specifications of I, O, and p for each object use the parameters INPUT, OUTPUT, and PROCESSOR, respectively. To form a dataflow graph, the objects in the workload must be connected by flows. A flow is specified implicitly using the OUTPUT parameter of the source object and the INPUT parameter of the destination object. Each INPUT, OUTPUT pair denotes a separate connection between the objects. In software specification languages such as ESML [9], the Ward/Mellor transformation schema [65], and PSDL [48], all transformations must be connected by flows to other objects. A transformation that does not receive data from other objects is useless; it can not do useful work. In contrast, in the SW, no transformation does useful work. All transformations only *behave* like they are doing something useful; they do not operate on real data. Therefore, we do not require that a transformation be connected to other objects. In some cases, the user may want to define a task that executes independently of other tasks. An example of this case is when specifying

SWs to study scheduling algorithms without considering task interactions. The workload would consist of a number of independent tasks whose only workload characteristic was the amount of CPU time required. For this case we do not require that the INPUT and OUTPUT parameters be defined. However, in most cases the user will be interested in the effects of task interactions on system performance. Hence, most transformations will be connected to others and will have INPUT and OUTPUT parameters assigned to them.

The two parameters for flows are flow type and the size of the data elements which pass along the flow. Because of the model construction rules (see Section 3.3.6) we can always determine the element size for a data flow from the components that it is connecting. A data flow must be attached at one end to either a store or a terminator, each of which has an ELEMENT_SIZE parameter. Hence, the flow can inherit this parameter from the object. Event flows carry no data, and thus require no size specification. Since the data element size can always be determined for a flow, we only need to be able to specify the type of the flow in the SWSL specification. Since flows only have a single parameter, it would be cumbersome to require that each flow be specified using the object definition notation which is used for the other components. Thus, we include the flow type into the specification of the INPUT and OUTPUT parameters for objects. Hence, the INPUTS and OUTPUTS serve a dual purpose. They declare the inputs and outputs of the object and define the flows that are attached to them. The format for the value of these parameters is

```
<connected object> :  <flow_type>.
```

The flow types are DISCRETE, CONTINUOUS, and EVENT, corresponding to intermittent data flows, continuous data flows, and event flows, respectively.

Object templates

An object template is specified by using the PROCESSOR parameter in conjunction with the INPUT and OUTPUT parameters. The PROCESSOR parameter defines the processor(s) to which the object is assigned. By providing multiple values for the parameter, the user specifies that an instance of an object is to be assigned to each of a number of processors. Each instance of the object will be assigned the same π_i parameter values. The INPUT and OUTPUT parameters will differ depending on the objects to which the copies of the object are connected. The connectivity of these objects is defined by using a special syntax for the INPUT and OUTPUT parameters. In the following discussion, the object for which the inputs and outputs are being defined will be referred to as the *current object*. The objects to which it is connected by flows from the inputs and outputs will be referred to as the *connected*

objects.

Transformation inputs and outputs are mapped one-to-one and in-order to function inputs and outputs (see the discussion of the functions file below). Therefore, the order in which inputs and outputs are specified is important. Furthermore, since SWSL supports object templates, the connected object may be one of many instances of an object. Hence, to accurately specify the connected object for an OUTPUT, for example, requires four pieces of information: the name of the connected object, the processor on which it is located, which INPUT of the connected object defines the other end of this flow, and the flow type. Since the current object specification may also be a template, the specification of inputs and outputs is somewhat complex. However, simple, compact specifications are possible in most cases. The formats for INPUT and OUTPUT specifications are as follows.

<object name> [
INPUT = <object$_{1,1}$> : <flow type$_{1,1}$> | ... | <object$_{1,n}$> : <flow type$_{1,n}$> ;
OUTPUT = <object$_{2,1}$> : <flow type$_{2,1}$> | ... | <object$_{2,n}$> : <flow type$_{2,n}$>;
PROCESSOR = <processor name$_1$>, ... , <processor name$_n$>;
];

This specification shows a single INPUT and a single OUTPUT within the context of an object definition. Multiple INPUTs and OUTPUTs can be defined by using multiple INPUT and OUTPUT statements. Each <object$_{i,j}$> specifies to which object the current object is connected by this INPUT or OUTPUT. The connected object specification has the following syntax.

```
<object name>.<processor>.<io number>
```

<object name> is the name of the connected object. <processor> is the name of a processor on which the connected object is located. The <io number> indicates to which INPUT or OUTPUT of the connected object this OUTPUT or INPUT is connected, respectively. INPUTs and OUTPUTs are numbered separately, beginning at 1.

If the current object is a template, vertical bars (|) separate specifications for the same INPUT or OUTPUT on different copies of the object. These specifications correspond positionwise to processor specifications in the PROCESSOR parameter list. That is, the jth entry in each INPUT and OUTPUT list is the specification for that INPUT or OUTPUT on the copy of that object located on the jth processor listed in the PROCESSOR list.

A simplified version of the INPUT and OUTPUT specification syntax may be used in most cases. The SWG can infer the connections given the simplified syntax. This specification scheme may be made clearer by looking at several examples.

In the simplest case, the INPUT and OUTPUT parameters describe the connections between nontemplate objects, as illustrated in Figure 4.1. This figure shows the object task

```
task [
INPUT = thermometer : DISCRETE;
OUTPUT = adj_temp : EVENT;
OUTPUT = temperature : DISCRETE;
PROCESSOR = rtcl11;
];
```

Figure 4.1: Example of simple INPUT and OUTPUT specification.

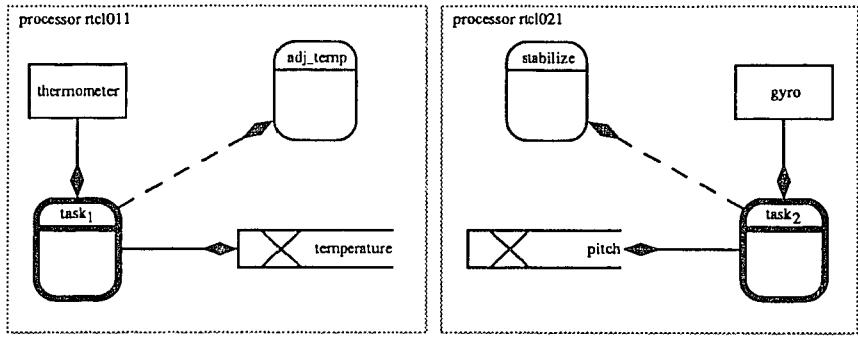
which is located on a processor labeled rtcl11. Its input is a discrete flow from thermometer and its outputs are an event flow to adj_temp and a discrete flow to temperature. The connected objects are not templates, so the name is sufficient for identification.

In the next case, the current object is a template and is connected to nontemplate objects. Identical copies of the object are to be placed on each processor listed in the PROCESSOR parameter list. There are two subcases to consider. In the first subcase, all copies of the object are to have the same INPUT and OUTPUT specifications. Then, it is sufficient to specify the INPUTs and OUTPUTs as in Figure 4.1. The SWG will copy the INPUT and OUTPUT specifications to all copies of the object. Note that the connected objects would have to specify connections to each copy of the current object.

In the second subcase, the INPUTs or OUTPUTs differ between the copies, as shown in Figure 4.2. The instances of the object template are highlighted. The user must specify different INPUTs and OUTPUTs for each copy. This example shows task as an object that has copies executing on both processors rtcl11 and rtcl21. The copy on rtcl11 has the same INPUT and OUTPUT specifications as the object in Figure 4.1. The copy on rtcl21 has its input from gyro and its outputs are stabilize and pitch. All of the π_i parameters would be exactly the same on both copies of object.

The next case involves specifying templates for objects whose instances are connected. This case is shown in Figure 4.3. In this case, the name of the connected object must specify the processor where the specific connected object is located. In this example, the output of the copy of task1 on rtcl11 is the copy of task2 on rtcl31. The output of the copy of task1 on rtcl21 is the copy of task2 on rtcl41. The INPUT specification of task2 reflect the connection from the viewpoint of task2.

It is assumed that the relationship between task1 and task2 in this example will be typical for object templates. That is, the connection between the first instances of the objects, i.e., task1 on rtcl11 and task2 on rtcl31, will hold for all subsequent instances of the objects, i.e., task1 on rtcl21 will be connected to task2 on rtcl41, etc.



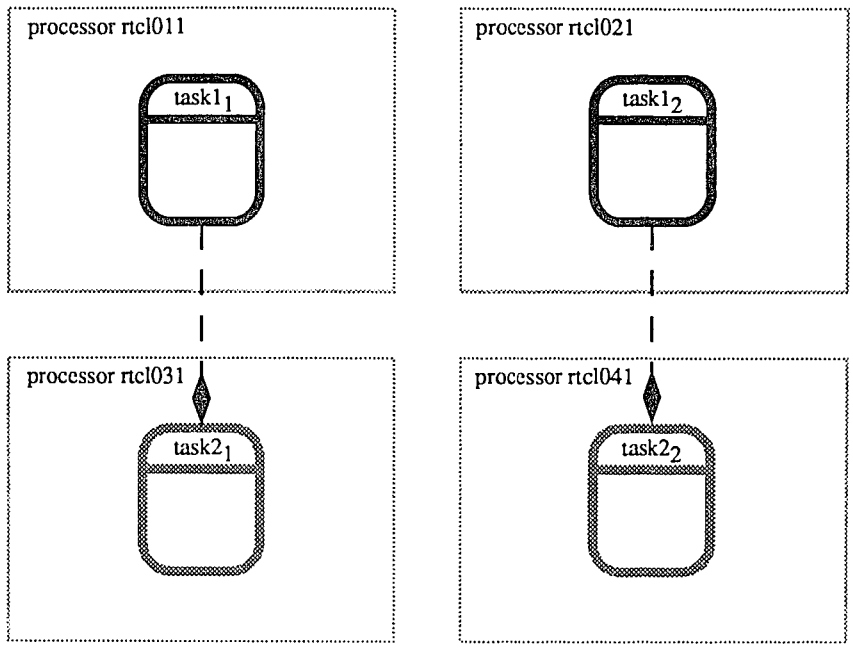
task [
INPUT = thermometer : **DISCRETE** | gyro : **DISCRETE**;
OUTPUT = adj_temp : **EVENT** | stabilize : **EVENT**;
OUTPUT = temperature : **DISCRETE** | pitch : **DISCRETE**;
PROCESSOR = rtcl11, rtcl21;
];

Figure 4.2: An object template with different INPUT and OUTPUT specifications for each instance.

Hence, there is a simplified notation that may be used in this case. Figure 4.4 shows an alternate specification of the connection between task1 and task2 that is equivalent to the specification in Figure 4.3. This notation allows quite large SWs to be defined with a very compact specification. A subgraph of nontemplate objects is defined and then processor labels are added to the PROCESSOR parameters for all the objects in the subgraph to produce a subgraph of templates. The subgraph of templates specifies multiple, identical subgraphs, but the objects in the subgraph will have different processor assignments. The INPUTs and OUTPUTs need not be altered in the specification.

In general, if there are more processors defined for an object (say object1) than there are |-separated entries in an INPUT or OUTPUT specification, then the SWG will fill in the remaining entries in the following manner. If the jth entry in an INPUT or OUTPUT list is a flow to object2 on processor $\texttt{proc}_i$, where $\texttt{proc}_i$ is the ith entry in the PROCESSOR parameter list of object2, then the $j+1$ entry of the INPUT or OUTPUT list will be to object2 on processor $\texttt{proc}_{i+1}$ if $i < n$ or $\texttt{proc}_i$ if $i = n$.

Next, we must handle the case where two objects are connected by multiple flows of the same type. The default is for the SWG to map the INPUTs to OUTPUTs of each flow type in the order of occurrence. However, in some cases one wants to change the order of the mapping. This may be required to connect properly with the INPUTs and OUTPUTs

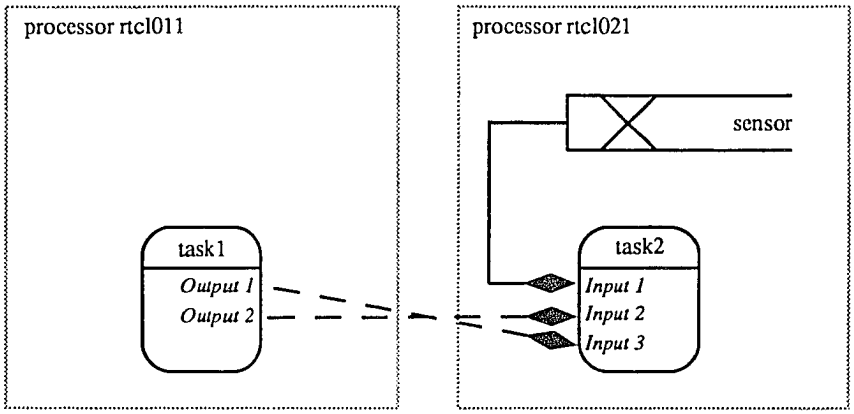



```
task1 [
OUTPUT = task2.rtcl31 : EVENT | task2.rtcl41 : EVENT;
PROCESSOR = rtcl11, rtcl21;
];

task2[
INPUT = task1.rtcl11 : EVENT | task1.rtcl21 : EVENT;
PROCESSOR = rtcl31, rtcl41;
];
```

Figure 4.3: Object templates for connected objects, complete specification.

```
task1 [
OUTPUT = task2 : EVENT;
PROCESSOR = rtcl11, rtcl21;
];

task2[
INPUT = task1 : EVENT;
PROCESSOR = rtcl31, rtcl41;
];
```

Figure 4.4: Object templates for connected objects, simplified specification.



task1 [
OUTPUT = task2.rtcl21.3 : **EVENT**;
OUTPUT = task2.rtcl21.2 : **EVENT**;
PROCESSOR = rtcl11;
];

task2[
INPUT = sensor.rtcl21 : **DISCRETE**;
INPUT = task1.rtcl11.2 : **EVENT**;
INPUT = task1..1 : **EVENT**;
PROCESSOR = rtcl21;
];

Figure 4.5: Specification of IO numbers.

ordering imposed on the transformation by its function. To accomplish this goal, we make use of the `<io number>` field in the `INPUT` or `OUTPUT` specification. The `<io number>` is the index of the `INPUT` or `OUTPUT` of the connected object. The indices begin at 1 and are numbered separately for `INPUT`s and `OUTPUT`s. An example of this notation is shown in Figures 4.5. This example shows three things. First, the inputs of `task2` are numbered starting at 1. Therefore, the first output of `task1` maps to input 3 of `task2`. Second, the `OUTPUT` assignments for `task1` are specified to "cross over"; the first output event from `task1` is the second input event for `task2` and the second output event from `task1` is the first input event for `task2`. Third, as shown in the third `INPUT` definition for `task2`, the processor value for `task1` need not be specified if it may be determined as described above.

Parameter Name	Description
TIMING	Defines whether the time limit should be used to terminate the experiment. Value is either TRUE or FALSE.
TIMELIMIT	Defines the maximum duration of the experiment.
SEED	Defines the initial seeds of the random number generator streams.
SEED_RESET	Contains one value for each value in the SEED parameter list. Denotes if the corresponding random number generator is to be reset to the specified seed value at the beginning of each run. Value is either TRUE or FALSE.

Table 4.4: Experiment parameters.

4.3.2 Experiment file

The experiment file defines the experimental parameters. It is divided into two sections: the CONSTANTS section and the PARAMETERS section. In the CONSTANTS section of the experiment file, the user may set the value of the Runs constant. The Runs constant defines the number of runs for the experiment. It defaults to 1, but the user may assign it a higher value to produce an SW with multiple runs. The PARAMETERS section contains one entry for each processor on which the SW is to execute. The entries are of the form

<processor name> [
<parameter list>
];

If the experiment parameters are to be the same for multiple processors, the DEFAULT entry may be used. All processors for which there is no explicit entry are assigned the default parameter values. The experiment parameters are listed in Table 4.4.

The experiment parameters are divided into two groups. The first group specifies the temporal characteristics of the experiments. These parameters indicate whether the experiment is to be timed and define the duration of the execution interval. The second group of parameters provide control of the statistical properties of the stochastic behavior of the SW. They define the seeds of the random number generators. Multiple random number generator streams may be defined. Each distribution function in the graph and function files can calculate its values from a separate stream. In this way, consecutive values generated by the distribution will be independent. The result is statistically independent events in the SWs execution. The SEED_RESET parameter may be used to reset the seed values at the beginning of each run. In this way, the behavior of individual streams may be reproduced.

```
<function name> {
  <input and output definitions>
  BEGIN
    <function code>
  END;
};
```

Figure 4.6: Function definition format.

INPUT = <input name> : <flow type> ;
OUTPUT = <output name> : <flow type> ;

Figure 4.7: `INPUT` and `OUTPUT` specification in the function file.

4.3.3 Functions file

This file defines the ϕ functions for the transformations. The simple operations and control constructs from the workload model have been adapted for specifying SWs. Simple operations are implemented using synthetic operations. Synthetic operations exercise specific resources in a predefined manner. The use of synthetic operations has been described in [4, 64, 60]. Special control constructs are defined to reproduce the data-dependent effects on the behavior of the looping and branching constructs in the workload model.

The functions file contains `EXTERNS`, `CONSTANTS`, and `CODE` sections. The code for the functions is defined in the `CODE` section. Function definitions follows the format shown in Figure 4.6. `<input and output definitions>` is a list of definitions of the input and output flows given in the form shown in Figure 4.7. These flows are given symbolic names that are used within the function. At compile time, the symbolic name is mapped to the corresponding `INPUT` or `OUTPUT` of the transformation that executes the function. The code that is executed by the function is defined between the `BEGIN` and `END` keywords. During each periodic or sporadic invocation of the transformation, the code between the `BEGIN` and `END` keywords is executed exactly once.

Computation and communication are implemented with synthetic operations. The synthetic operations are located in a library of operations. Synthetic operations are implemented as C functions. These functions are parameterized so the user can control their behavior. By defining them as functions, we hide the implementation details. Hence, the SW function specifications are made target system independent. All system dependencies

```
LOOP <expr>
{
  <function code>
};
```

Figure 4.8: LOOP construct.

are contained in the implementation of the operations.

We have collected a number of synthetic operations for the library. Some of these were taken from the publicly available Bell Labs Benchmark suite and the dhrystone benchmark. They perform functions such as Ackerman's function, floating point arithmetic, and word counts which exercise specific system functionalities. The operations are parameterized to control the number of iterations of each function. We also include a function to exercise memory by producing reference strings based on the algorithm in [2]. Additional operations may easily be added to the library.

Of particular importance are the `sread` and `swrite` operations. They are defined to be generic input and output operations. The primary parameter for these functions is the symbolic name of the input or output. The user need not specify any information about the object with which the function is communicating via the operation. The operations take information generated by the SWG for each input and output of the function and determine the appropriate system call(s) on the target system to use to perform the appropriate reading or writing operation. Therefore, they may be used in any function, regardless of the transformation which executes it and regardless of the objects to which the transformation is connected. These operations increase flexibility by introducing the capability for *plug-in* functions. During normal use of SWSL and the SWG, it is anticipated that the user will code a number of functions with different behaviors representing different types of tasks. These functions will be "plugged in" to transformations as needed for the particular application. Thus, SWs can be quickly constructed from components with known characteristics.

Control flow within a function is achieved by using two constructs, `LOOP` and `SWITCH`. `LOOP` is an adaptation of the `while - do` loop in the workload model. It is a single entry, single exit looping construct. The format for the `LOOP` construct is shown in Figure 4.8. The parameter `<expr>` specifies the loop count. The `LOOP` may be made to execute a constant number of times for each run, or it may execute a random number of times by specifying a distribution function as the loop count. By looping a random number of times, the loop simulates the behavior of data-dependent loops in the workload being modeled.

```
SWITCH
{
  percent_1 :
  {
    <function code>
  };
  percent_2 :
  {
    <function code>
  };
  ⋮
  percent_n :
  {
    <function code>
  };
  remaining :
  {
    <function code>
  };
};
```

Figure 4.9: SWITCH construct.

Branching is accomplished using the SWITCH construct. The SWITCH construct is shown in Figure 4.9. It is a generalization of the `if-then-else` construct in the workload model. It is derived from the SWITCH operation in [60]. In the SWITCH statement, the user specifies alternate blocks of code to be executed. Probability values are assigned to each block. Each time the SWITCH is executed, one block is selected at random. By branching probabilistically, it simulates the behavior of real applications that branch based on data values. The $\texttt{percent}_1, \ldots, \texttt{percent}_n$ values, where $\sum_{i=1}^{n} \texttt{percent}_i \leq 100$, represent the percentage of times that each corresponding branch will be taken. For example if $\texttt{percent}_1 = 60$ and $\texttt{percent}_2 = 10$ then 60 percent of the branches will be to the first block of code and 10 percent of the branches will be to the second block of code. The `remaining` label indicates that the remaining percentage (up to 100) of the branches execute its block of code. If the percentages do not add to 100 and there is no `remaining` case, then the remaining percentage of the time no operations are performed.

C code may be inserted at any point in the CODE section using a `verbatim/endverbatim` block. This C code is copied directly to the C code being generated for the function by the SWG. An SWSL function may contain any combination of synthetic operations and user code.

4.4 Synthetic Workload Generation

We have designed and implemented the SWG which compiles SWSL. The SWG completely automates the generation of SWs. The synthetic workload generation process is shown in Figure 4.10. The SWG compiles the SWSL graph file to produce an internal representation of the task graph. It checks the graph for compliance to the connection rules. It then processes the inputs and outputs of the components to expand any specifications that use the simplified specification notation. Next, it compiles the experiment file and generates arrays containing the experimental parameters for each processor. Then, it compiles the functions file and produces C language code for each function of each transformation on each processor. While producing these files, it uses information from the task graph to expand the input and output symbolic names in the functions. The parameters of the object to which the flow is connected is specified in the graph file. Therefore, for each input and output specification which is compiled, the SWG generates a data structure which contains the important parameters of the object to which the input or output flow is connected. This information may be used by the synthetic operations within the function. For all other synthetic operations, the SWG copies the call directly to the C code of the function being defined. Input and output operations which operate on flows specify the flow name as a parameter. The flow name is expanded by the SWG to a reference to the proper data structure before being copied to the function C code.

Next, the SWG generates files containing arrays of the parameter values for the objects on each processor. These arrays are described in Chapter 5. Finally, the SWG uses the processor assignment information from the graph file to direct the *make* utility to compile the generated code for each function. The compiled function files are then linked with the library of synthetic operations, the library of distributions, and the SW driver[3] to create an executable image. The executable images for all the processors compose the SW. They may then be downloaded to the target system and executed.

4.5 Summary

SWSL is a language for specifying synthetic workloads for distributed real-time systems. It is designed to be easy to use while providing compact and flexible specifications. It is based on a workload model which makes it compatible with commonly used software specification notations. Hence, it is capable of accurately modeling real workloads. The

[3]The SW driver is described in Chapter 5

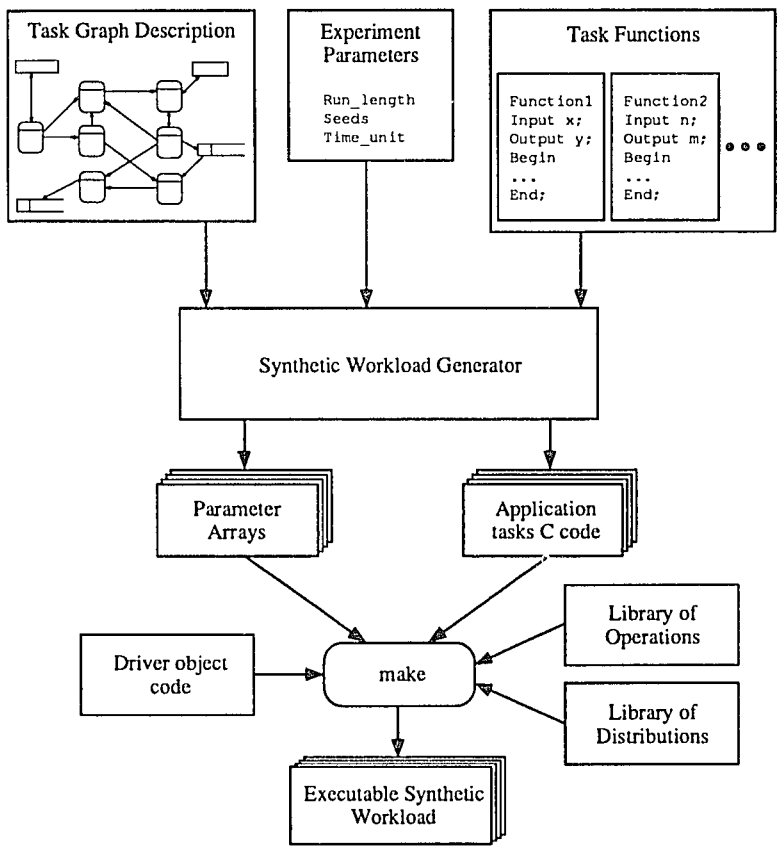


Figure 4.10: Synthetic Workload Generation.

language includes facilities for experiment support such as support for multiple runs, object templates, multiple random number generator streams, and modeling of stochastic behavior. The SWG has been fully implemented. It supports all features described in this chapter. Its use is demonstrated in Chapters 6 and 7.

CHAPTER 5

THE SYNTHETIC WORKLOAD

5.1 Introduction

The synthetic workload (SW) executes on a distributed real-time system. The SW on each processor is composed of a set of processes that can be divided into two groups: the *synthetic application tasks*[1] and the *driver processes*. The synthetic application tasks implement the user-specified SW. The driver controls the SW in the context of an experiment.

The use of a driver to control an SW is not a new concept. A good discussion on workload drivers may be found in [24]. Our contribution is to describe the important features of a driver for SWs to be used in distributed real-time systems. These features have been implemented in the driver for our SW.

Our SW uses distributed control because centralized control would cause additional load on the system's communication facilities. This overhead would adversely affect the performance indices being measured. Distributed control allows the SW to execute without imposing any overhead load on the communication system. The only communication overhead is caused when the distributed SW control elements synchronize at the beginning and end of each run. Synchronization is necessary if synthetic tasks on different processor require synchronous communication. It is especially important if the tasks are periodic. Synchronous communication between unsynchronized, periodic tasks on different processors can cause intolerably long waits for the task which is invoked first as it waits for the other task to be invoked. The SW is not like a distributed discrete event simulation. In distributed simulations, simulation tasks execute in simulated time. If some processors start later than others, then the algorithms which maintain global consistency of simulation time can quickly bring the processors into relative synchrony. In the SW, all tasks run in real

[1]The term "task" is used to distinguish the synthetic application processes from the driver processes.

("wall clock") time. The execution cycles of periodic tasks are fixed and are independent of the behavior of the periodic tasks. They will not become synchronized through the actions of the tasks. Explicit synchronization is required.

The SW implements all the experimental support provided by SWSL. It fully supports the multiple run facility. The duration of each run is determined by the `TIMELIMIT` parameter. Between runs, the workload is completely reset. It is then recreated with the parameter values for the next run. The SW is implemented to work in harmony with monitors and other measurement mechanisms. It presents itself to the system as an application program. Therefore, existing monitors may be used without modification.

Real-time systems control physical processes. For some applications, the system can not be evaluated while connected to live sensors and actuators. This is especially true when using an SW, which can not properly interpret data from sensors nor can it send the proper commands and data to actuators. For this reason, the SW simulates the input and output properties of terminators. Our implementation of simulated terminators is an expansion of external event generation described for [60] and various techniques for device simulation, e.g., [31].

In addition to these behavioral features, the SW has implementation features to make it easier to use and easier to port to a new system. Although the synthetic tasks may change for each evaluation, the driver has a fixed structure. It is compiled once for a given target system. After that, it is only necessary for the SWG to link the driver's object code with the SW. This separation between SW tasks and the driver makes it faster to compile SW specifications because the driver code does not need to be recompiled for each new specification. Also, the driver and the library of operations contain the system-dependent code for the SW. Porting is simplified by localizing the system dependencies.

In section 5.2, the structure of the SW is outlined, and its implementation and use are described. As we describe the SW, we will discuss how the above features are implemented. In Section 5.3, we discuss the overhead incurred by the driver. The chapter concludes with Section 5.4.

5.2 The Synthetic Workload

The graphical representation of the SW for each processor is shown in Figure 5.1. The driver processes and data structures are shown in the labeled box. All other transformations, stores, and terminators represent the user specified SW.

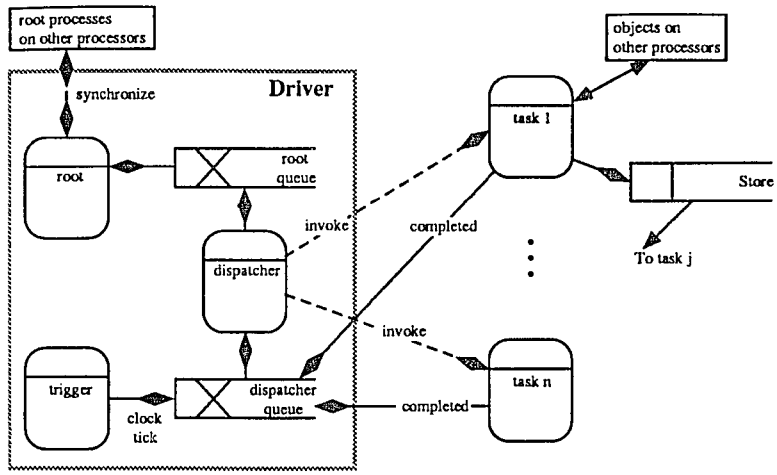


Figure 5.1: Data flow model for the SW processes on a single processor.

5.2.1 Synthetic Application Tasks

The synthetic application tasks are responsible for generating the resource demands on the processor. They are the implementations of the transformations in the SWSL specification. Their structure has been discussed in Chapter 4, and will not be discussed further here.

5.2.2 Driver Processes

The driver controls the execution of the SW in the context of an experimental evaluation. The driver processes are responsible for initializing and starting the SW, for the synchronization between the SWs on the various processors, and for the scheduling of stochastic events and simulated I/O.

The *root* process executes first at system initialization. It spawns all the other driver processes and synthetic application tasks and creates the data structures that implement the stores. It also synchronizes with the root processes on the other processors at the beginning and end of each execution. This synchronization is required if the SW tasks participate in synchronous communication.

The driver uses the `TIMEOUT` parameter from the experiment file to specify the maximum time that each run is to execute. When this time is reached, the run ends. At the end of a run, all SW tasks, data structures, and processes, except for the root process, are deleted. No history information is kept between runs. Hence, runs are statistically independent. After the execution of each run, the root process waits for input from the

user. This wait gives the user time to upload performance data or reset measurement instruments before the system is reinitialized for the next run.

The two other driver processes are the *trigger* and *dispatcher*. The trigger acts as a software timer that periodically sends clock tick messages to the dispatcher. The dispatcher counts time and performs a number of time related functions. For both of the implementations used for the evaluations in Chapters 6 and 7, these functions included implementing scheduling of periodic tasks and the enforcement of task deadlines. On a system whose operating system fully supported periodic tasks, these functions of the dispatcher would not be necessary. Therefore, scheduling of periodic tasks and enforcement of deadlines are not considered to be permanent parts of the SW. The dispatcher is also used to simulate the operations of terminators.

5.2.3 Software Structure

The SW software is structured to be modular, and the code that may be altered by the user is separated from the fixed code. The only time the code should be altered is when it is being ported to another target system. Once installed, it should not require further modification.

Data Structures

The SWG produces parameter arrays that define the workload parameters from the SWSL graph file. These arrays are read by the root process to create the appropriate tasks and data structures for the specified SW. The parameter arrays contain an array for each parameter of each object type. Each is an $n \times m$ array, where n is the number of objects defined of the specified type and m is the number of runs. For example, consider the array

```
int trans_priority [2][3] = {
{ 10, 5, 12},
{ 15, 2, 12},
};
```

that defines the values for the PRIORITY parameter for transformations. This array indicates that there are two transformations in the workload and values have been defined for three runs. The array name includes an indication of both the object type and the parameter. This naming scheme is necessary since some parameter names are shared by multiple objects, e.g., the TYPE parameter for both terminators and stores.

Some parameters may take on values that are generated at run-time by calling a function. An example is the SPORADIC parameter for transformations. The value of this parameter is specified in SWSL as a call to a random distribution function. At run-time, the distribution function is called and the value returned is assigned to the parameter. Some of these functions require parameters; some do not. Furthermore, the values of the function's parameters, or even the selection of which function to execute may change between runs. The SW driver must call these functions. Since the driver has a fixed structure and is precompiled, it can not know about the names of the functions and their parameters. Therefore, we have added one level of indirection in the calling sequence for these functions. For each function, $f(x)$, the SWG defines a uniquely named function, $f'() = f(x)$, that calls the required function. For each $f(x)$ there is a unique function defined for each run. The functions are named

<transformation name>_<parameter name>_<run number>

indicating that the named transformation executes the function to obtain a value for the parameter during the specified run. These functions return the value that is calculated by the user-specified function. We clarify the definition of these functions with the following example.

Suppose the workload consists of a single transformation, task1, which is to be invoked sporadically, and the time between invocations is defined by a geometric distribution. There are two runs in the experiment. In the first run, the mean of the sampling distribution is to be 10, and in the second run it is to be 15. Then the following functions will be defined.

```
extern int geom();
int task1_sporadic_1 ()
{
  return(geom(10));
};

int task1_sporadic_2 ()
{
  return(geom(15));
};
```

The first entry defines task1_sporadic_1() as a function that returns the value returned from the geometric distribution function with a mean of 10. The second entry defines task1_sporadic_2() as a function returning the value taken from a geometric distribution with mean 15. The corresponding table for the SPORADIC parameter for transformations will be

```
int (*trans_sporadic [1][2])() = {
{ task1_sporadic_1, task1_sporadic_2},
};
```

Using this technique, the SW driver can obtain the proper value for the SPORADIC parameter by executing the function in the array.

The sizes of the parameter arrays change between experiments as the number of objects and the number of runs change. Hence, the source code for each function that accesses these arrays must be recompiled for every change to the workload. We wish to reduce the time required to recompile the SW between experiments. Therefore, we define an *current-run array* that contains only the values for the parameters for a single run. During the execution of the SW, this array will contain the parameter values for the run being executed. The proper values are placed in the current-run array by the initialization function which is described below. All other driver functions access only the current-run array. The initialization function is located in a separate source file. It is the only driver source file that must be recompiled between experiments. A similar scheme is used for the arrays containing the experimental parameters.

The Root Process

The pseudo-code algorithm for the root process is shown in Figure 5.2. The root process is responsible for creating all the user-specified components of the SW. Once the components have been created and spawned, the root process waits for the SW to complete execution. This process is contained in a loop that executes once for each run.

Two functions are called by the root process. The first is the `initialize()` function. It is called at the beginning of each run. It resets the driver data structures and places the appropriate parameter values from the parameter arrays into the current-run array.

The second function called by the root process is the synchronization function. In general, it can be assumed that the target system maintains synchronized clocks on the processors. Synchronized clocks are necessary if the system is to provide guarantees for time-constrained communication between processors [39]. In the presence of synchronized clocks, the synchronization function is simple. The SW on one processor would be designated as the master. At the beginning of a run, it broadcasts a message to the other SWs indicating the start time. The start time is calculated to be a time at a sufficient distance into the future to allow all drivers time to receive and process the message. Upon receipt of the message, each SW waits until the start time, as indicated by its local clock. All clocks reach this time

```
root()
{
  for each run
  {
    initialize() ;
    create(dispatcher queue);
    create(root queue);
    for each store
    {
      create data structure for the store;
    }
    for each terminator
    {
      create terminator data structure;
      create terminator process;
      schedule the START activity for the terminator;
    }
    for each transformation
    {
      spawn the task;
      schedule the START activity for the task;
    }
    synchronize();
    spawn(dispatcher);
    spawn(trigger);
    wait(root queue);
    synchronize();
    delete all tasks, processes, and data structures;
  }
}
```

Figure 5.2: Root process structure.

simultaneously, and the SW is synchronized. In a system without synchronized clocks, the other drivers begin executing as soon as they receive the message from the master. This technique results in much looser synchronization due to the different times at which the drivers receive the messages.

The first time the `synchronize()` function is called within the loop is to ensure that the root processes on all processors are at the same point in their execution. All processors then perform their initialization and begin execution of the application tasks. Once each processor completes its run, it synchronizes with the master processor. When all processors have completed, the master notifies the user, who may then reset any measurement mechanisms or upload performance data from the processors. The master then waits for a predetermined period of time or for input from the user console; either mechanism may be used. When the wait period has elapsed or the command from the user is received, the master begins the next run.

The Trigger Process

The trigger process periodically sends clock tick messages to the dispatcher data structure. Once it has sent a message, it sleeps for a length of time specified by the `TIME_UNIT` parameter. This user-specified parameter determines the basic unit of time (a multiple of the system's clock period) that will be used for all time-related SW actions.

The Dispatcher Process

The dispatcher uses the trigger messages to count time. The time value is used for the scheduling of activities. The dispatcher maintains an activity[2] queue which is similar to event queues used in discrete event simulation systems. Activities correspond to actions that are to be performed by the dispatcher at specific times. Activities indicate each task's start time, as defined by the `START_TIME`, `PERIOD`, and/or `SPORADIC` parameters, and deadline, as specified by the `DEADLINE` parameter. There are also activities to indicate the times when the simulated terminators are to produce or consume data.

The dispatcher uses the `RATE` parameter for terminators to determine when to send messages or events from source terminators and when to read messages or events at sink terminators. The SW simulates terminators by using a data structure and a task for each terminator. The data structure is used if the simulated terminator is to produce or receive data. Synthetic application tasks that communicate with the terminator will send and

[2]The term "activity" is used to avoid confusion with events in the workload model.

```
function()
{
  loop forever
  {
    wait for "invoke" signal from dispatcher;
    do_work;
    send "completed" message to dispatcher;
  }
}
```

Figure 5.3: Function structure.

receive messages to/from the data structure. The terminator task is used if the simulated terminator is to produce or receive signalled events. When instructed by the dispatcher, it sends signals to the appropriate application tasks or reads signals sent by application tasks.

Since the dispatcher is being used for scheduling tasks, it receives other messages through the dispatcher queue besides clock ticks. Each task sends a message to the dispatcher when it has completed execution. The dispatcher uses this information to cancel the corresponding task deadline activity in the activity queue.

To support scheduling by the dispatcher, the application tasks generated by the SWG all have the same skeleton structure, which is shown in Figure 5.3. Each time the task receives an invoke signal from the dispatcher, the remaining code in the loop executes once. Between invocations, the task blocks waiting for the invoke signal from the dispatcher. The `do_work` section of the code represents the execution of the task ϕ function. Once its work is completed, the task must notify the dispatcher so that the dispatcher may cancel the deadline activity for the task.

5.3 Driver Overhead

As was stated earlier, the application tasks must produce the desired workload characteristics. It is their structure and behavior that may be changed. The structure of the driver is fixed; it's demands for resources are not controlled by the user. Obviously, the driver's behavior is influenced by the workload parameters. The dispatcher, for example, will execute more frequently in a workload with short period tasks than it will in a workload with longer period tasks. However, the amount of time that it executes per task invocation will be fixed. The driver thus produces a calculable overhead per task execution. This overhead may be determined and taken into account as the workload is being tuned for a

particular experiment. Since we are working with real-time systems, we want the driver overhead to be as low as possible.

We ran an example SW on HARTS in order to measure the driver overhead. The workload consisted of four application tasks. The execution of each task consisted of a loop to use CPU time. This workload was sufficient to measure the fixed per-invocation overhead. Data was collected for each of the three driver processes (root, dispatcher, and trigger). The dispatcher time was measured separately for each of the functions that it performed. These functions include: invoking a task, aborting a task which has reached its deadline, and processing a clock tick from the trigger. For each operation, we measured the time immediately after the dispatcher received a message from the dispatcher queue and dequeued the appropriate activity from the activity list until immediately before it made the system call to request the next message from the dispatcher queue. The time required for the dispatcher to receive a message was on the order of 100 μsec.; this time is not included in the following measurements. The average time to invoke a single task was 118 μsec. The time was clearly dominated by the approximately 100 μsec. it took to perform the system call to send the invoke signal to the task. The time to process a deadline event was 469 μsec. This time was due to the number of system calls that were performed to destroy the task and spawn and activate a new version of it.

The trigger process took 247 μsec. per trigger. The actual amount of code in the trigger process is small, but two system calls were made per trigger. The root process executed for less than 60 msec. This time was primarily used in system initialization. The root process was suspended while the rest of the workload was executing. Therefore, this overhead did not affect the performance of the SW.

5.4 Summary and Conclusions

In this chapter we described the executable SW. It is composed of the synthetic application tasks and the driver. The synthetic application tasks implement the user specified workload. The driver provides control of these tasks in the context of an experiment. We outlined a number of features which should be supported by a driver in a distributed real-time system.

Our driver implements these features. It synchronizes the distributed SW at the beginning of each run, reinitializes the SW between runs, and simulates the actions of terminators. It also provides task scheduling facilities, if needed.

The SW driver is similar to the driver for a discrete event simulation. It controls the

start and termination of an experiment. During the experiment it controls the execution of various activities or events. In fact, the dispatcher manages its activity queue using common techniques for managing event queues in simulations. The main difference between the two types of drivers is that the SW is not a simulation. The tasks in the SW are actually executing on the system in real time. The dispatcher must process activities at the correct time. Once an activity is processed, the dispatcher cannot move time ahead and process the next activity. Similarly, it must not wait too long to process the next activity. If the timing requirements are not met, the driver could cause incorrect behavior from the SW, an unacceptable situation. Another difference concerns what events are controlled by the driver. In a simulation, the driver processes all events. It handles all interactions between the simulated objects. The SW driver does not control all aspects of the SW. The SW may contain a wide range of control constructs and task interactions which are unknown to the driver. Sporadic tasks may be "invoked" by receiving data from other tasks without being scheduled by the dispatcher. Since for real-time systems we are generally concerned with the timing behavior of the workload, the SW driver must not produce a significant overhead. For tasks with periods of 10–20 msec., such as those described in Chapter 6, the driver overhead represents approximately two percent of each period. This value is acceptable.

CHAPTER 6

REPRESENTATIVENESS OF THE SYNTHETIC WORKLOAD

6.1 Introduction

This chapter addresses the problem of verifying that the SWG is capable of producing representative SWs. Our goal is to demonstrate that, given the proper SWSL specification and workload parameters, the SWG is able to produce an SW which accurately represents an actual workload. To demonstrate this ability, we first determine how to measure the representativeness of the SW. Then, we describe an actual experiment which used the SWG to produce an SW to accurately model an independently developed real-time workload.

Many performance analyses are concerned with determining how the system will behave under an actual or proposed workload. It is therefore important that the SWG be capable of generating SWs which accurately represent the actual workload. The problem of how to create a representative SW has been studied since the development of the first benchmark programs [35, 36]. In the case of benchmarks, the definition of representativeness was restricted to the ability of a benchmark program to model the behavior of a generalized example of a program from a given application domain. The problem was to define how to compare the behavior of the benchmark with that of an actual application program. Since that time, three models have evolved which may be used to determine if an SW is representative [29]. These are the *resource-usage model*, the *functional model*, and the *performance-based model.*

In the *resource-usage model* an SW is said to be representative of a workload if the jobs in the SW consume system resources at the same rate as the jobs in the workload. Since an SW is an abstraction of a real workload, it does not consume resources exactly like the real workload. However, it can exhibit the same resource consumption pattern or ratio between the consumption rates of different resources, e.g., the number of disk accesses per

unit of CPU time used. This model has been used frequently in workload characterization studies. For example, it was used by Domanski [18] to produce a representative synthetic workload based on an actual workload for a database management system from which resource usage data had been obtained. Using data from the log tape of the execution of an application, the jobs in the application were plotted in the space defined by the quantity of CPU time, input message length, output message length, and number of database calls used by the job. A clustering algorithm was applied to find representative jobs from the workload. The parameters of the representative jobs were then used as the parameters of the SW.

The second model is the *functional model.* The functional model states that an SW is representative of the real workload if it performs the same functions as the real workload. For example, an SW would be considered representative of a payroll processing workload if the SW contained only payroll processing jobs.

The functional model is much more system independent than the resource-usage model, because it specifies only the functions to be performed, not the manner in which they are to be performed. One need only specify which functions are to be considered representative, then construct a workload (from a smaller number of these representative tasks) which performs those functions in proportional amounts.

The *performance-based model* was first defined by Ferrari [23]. According to this model, an SW is representative if it causes the system to behave in the same manner as the real workload, as measured by some specified performance indices. This notion was formalized in [26]. Given a synthetic workload $\mathcal{W}'$, a workload $\mathcal{W}$, a set of performance indices $\mathcal{P}$, and a target system $\mathcal{S}$, then

Definition 1 $\mathcal{W}'$ *is* representative *of* $\mathcal{W}$ *if it produces the same values of the performance indices* $\mathcal{P}$ *as* $\mathcal{W}$ *when running on the same system* $\mathcal{S}$.

Each of these three models has its strengths and weaknesses. The main advantage of the resource consumption model is that it is intuitive. A workload can be thought of as a set of programs which consume resources in the system. This model translates readily into the design of an SW. The disadvantage of this model is that it is highly system dependent. The resources consumed by the workload are particular to the system on which it is executing. Likewise, the consumption patterns are determined in part by the configuration of the system.

The functional model has the advantage of simplicity. One need only choose workload components which perform the proper functions, and the result is a representative

workload. The main problem with this model is that there can be a wide variation in the ways that a given function may be implemented. Therefore, there is no guarantee that measurements of the system which are made while the SW is executing will be any indication of the values which would be obtained if the actual workload were being executed.

The advantage of the performance-based model is that it relates directly to the reason that we are using an SW. We are using an SW to aid in performance evaluation. As such, our only concern is how the SW affects the performance of the system. The other two models are useful for guiding the design of the SW, but the performance-based model is the only one which provides a quantitative measure of representativeness which can be related directly to the evaluation being undertaken. It is therefore the best choice for our purposes.

The performance-based model can be used at any level of abstraction. For example, suppose the SW is being used to evaluate the communication subsystem of a distributed real-time system. The measured performance index is the message delivery time at a certain traffic level. The traffic level is determined by the frequency of message generations, which is indirectly proportional to the time between message generations by the sending task. We are thus measuring two performance indices: the message delivery time and the task execution time between message generations. In an actual workload, this latter index would be determined by other factors such as the number of operations to be performed, number of memory accesses, and resource contention between tasks. However, since the goal of the evaluation is not to characterize the delays caused by these factors, the SW need not reproduce them individually. It need only reproduce the cumulative delay.

To use this definition of representativeness, we must perform experiments on a real system. To obtain $\mathcal{W}$, $\mathcal{P}$, and $\mathcal{S}$, we devised a realistic performance evaluation experiment such that $\mathcal{P}$ would be measured while $\mathcal{W}$ executed on $\mathcal{S}$. The experimental scenario is described below. We executed $\mathcal{W}$ on $\mathcal{S}$ and measured the values of $\mathcal{P}$. Then, an SW, $\mathcal{W}'$, was constructed to model $\mathcal{W}$. The $\mathcal{P}$ and $\mathcal{S}$ selected for the experiment were used to determine the representativeness of $\mathcal{W}'$ relative to $\mathcal{W}$.

In the experimental scenario, the real-time computing system, $\mathcal{S}$, was the computer used at the Robotics Laboratory at the University of Michigan to control a pair of robot arms. The workload, $\mathcal{W}$, was the software that interprets movement commands from the user and controls the movement of the arms. This workload was chosen for two reasons. First, it contained a real-time element and an element with no real-time requirements. Thus, the SW could model both periodic and sporadic (aperiodic) tasks. Second, it was developed independent of the SW by researchers who were unaware of the existence of the

SW. Likewise, the SW was developed independent of it. As such, its design was not biased toward structures and behaviors which might be easily modeled by the SW, nor was the SW biased toward being able to model its structure. It therefore served as a fair test of the SW's ability to model actual workloads.

We wanted to make this evaluation as realistic as possible. Therefore, to determine the performance requirements of the application and, therefore, the proper indices to use as $\mathcal{P}$, we consulted with the researchers who originally selected and purchased the system[1]. The primary performance requirement used in the selection processes was that the real-time portion of the workload must meet its timing requirements. The timing requirements are discussed in Section 6.3.1.

This chapter is organized as follows. In Section 6.2, we describe the target system $\mathcal{S}$, the robot control software, $\mathcal{W}$, and the representative SW. In Section 6.3, we describe the representativeness experiments. We define the experimental parameters, the data collection technique, and the performance indices, $\mathcal{P}$, that were measured. The results of the experiment are presented and analyzed in Section 6.4, and we conclude the chapter with Section 6.5.

6.2 Target System

The target system consisted of a multiprocessor computer controlling a pair of robot arms. In this section we will define the system architecture. We will then describe the robot control software and the representative SW.

6.2.1 System architecture

The robot control computer was composed of five processor cards with a variety of CPUs: two Motorola 68020s, two Motorola 68030s, and a Motorola 68040. The processor cards had their own local memory and were housed in a common card cage and connected on a VMEbus backplane. VxWorks was used as the real-time operating system for the processors. The system also included an analog-to-digital converter board, a digital-to-analog converter board, and an arm interface board on the VMEbus. These boards were used to communicate with the robot arms. The processors were connected to a workstation by an Ethernet. The computer controlled two Puma 560 industrial-grade robot arms. Each arm had six degrees of freedom and a gripper that could grasp objects. The robot control environment is shown in Figure 6.1.

[1] Primarily Joseph A. Dionese.

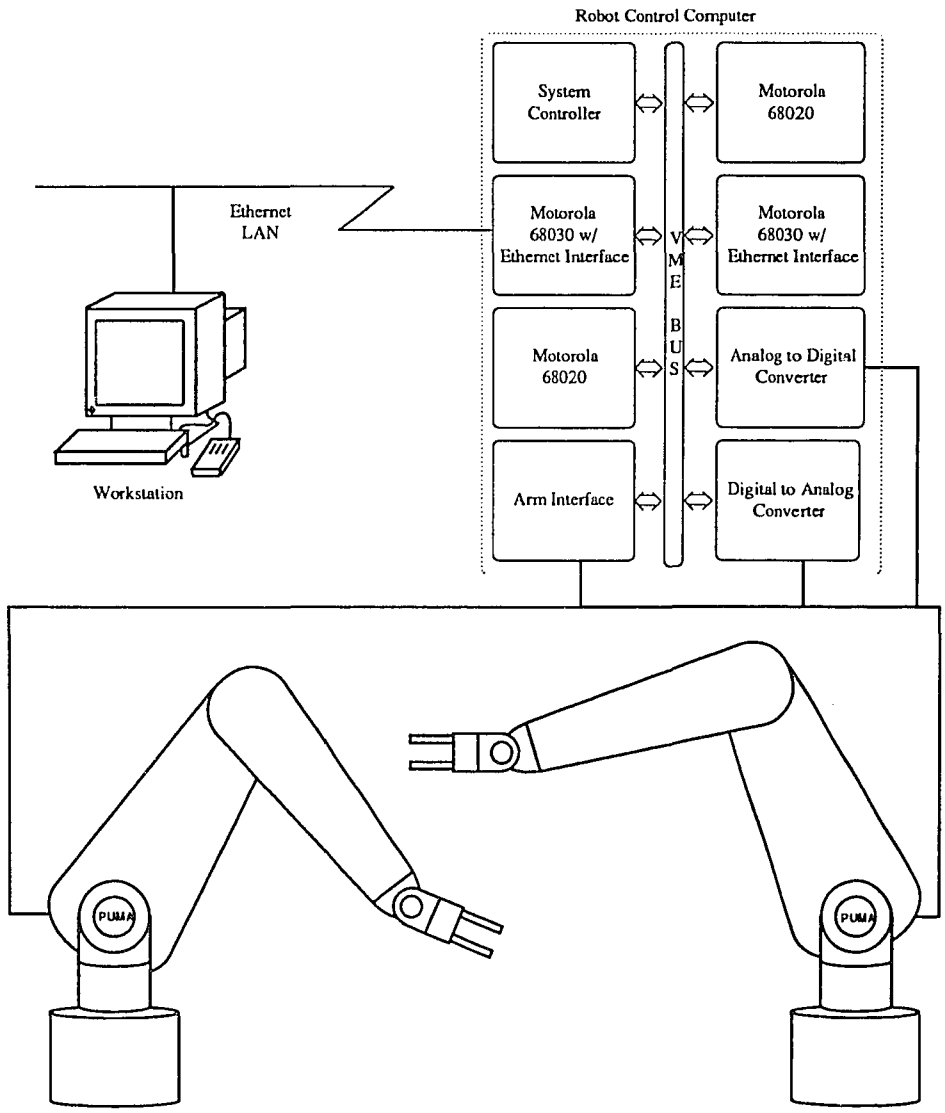


Figure 6.1: The robot control environment

6.2.2 The robot control software

The two robot arms were controlled independently by software on separate processors. There was no low-level coordination of the arms. Therefore, we only considered the software for a single arm executing on a single processor. The robot control software was composed of three tasks: a client, a server, and a level0 control task. The client task executed on a workstation which was separate from the robot control computer. The client task acted as the user interface to the control software. It processed user commands and passed them via a Unix socket to the server. The server task executed on the robot control computer. It accepted commands from the client and processed them accordingly. Many of the commands were executed directly by writing the appropriate values to the arm interface and digital-to-analog converter boards. The server also updated global tables to change the system status or the operating mode. Some commands were reformatted and placed in a message queue which was read by the level0 task. The level0 task ran as a periodic interrupt executing with a frequency of 60 Hz. It performed the control loop for the robot arm. During each invocation, the loop read the arm position sensors, calculated new positioning commands based on an adaptive control-law algorithm, processed commands from the server, and wrote the new motion commands to the arm interface board. A flowchart of the task's internal structure is shown in Figure 6.2. The level0 task was the primary task in the workload. The server and the level0 task executed on the same processor. The only other tasks which executed on the system were the VxWorks system tasks.

There were three primary operational modes for the robot control software. In **standby** mode, arm motion was stopped. This mode was used during initial calibration of the arm position sensors. In **torque** mode, arm movement was controlled by a high-level control module which calculated the torque values for the joints. This mode relieved the robot controller from generating the desired arm path and calculating the feedback law. For our evaluation, we were interested in evaluating the timing requirements of the application when it calculated the feedback law as part of its control loop. Therefore, this mode was not used. In **position interpolated** mode, the robot arm received motion and gripper commands from the user interface and moved and acted accordingly. This was the primary operating mode for the robot arm.

6.2.3 The synthetic workload

A uniprocessor SW was created to model the robot control software. Figure 6.3 shows the structure of the SW. This figure uses a simplified notation for clarity. Pairs of

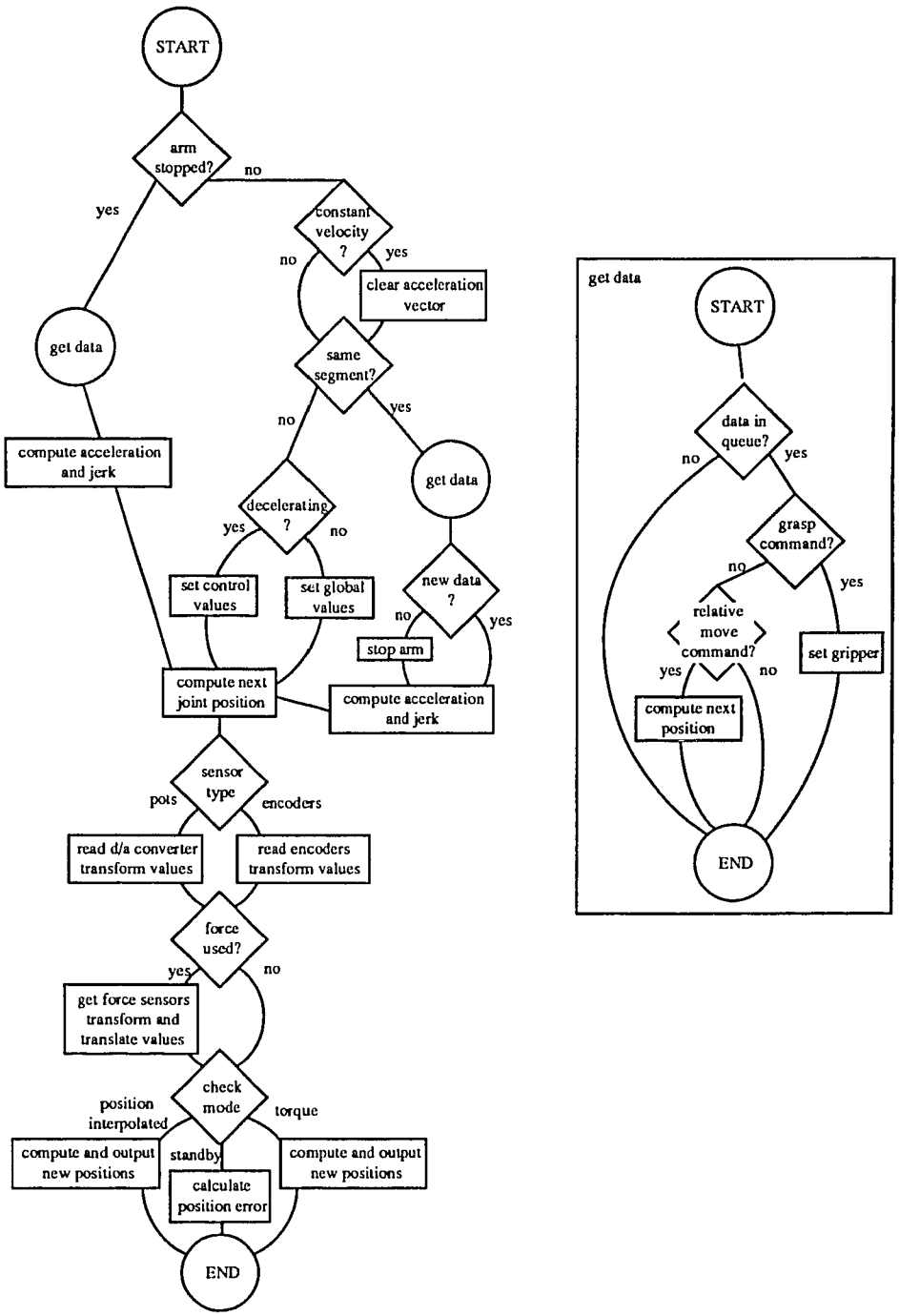


Figure 6.2: Flowchart for the level0 task

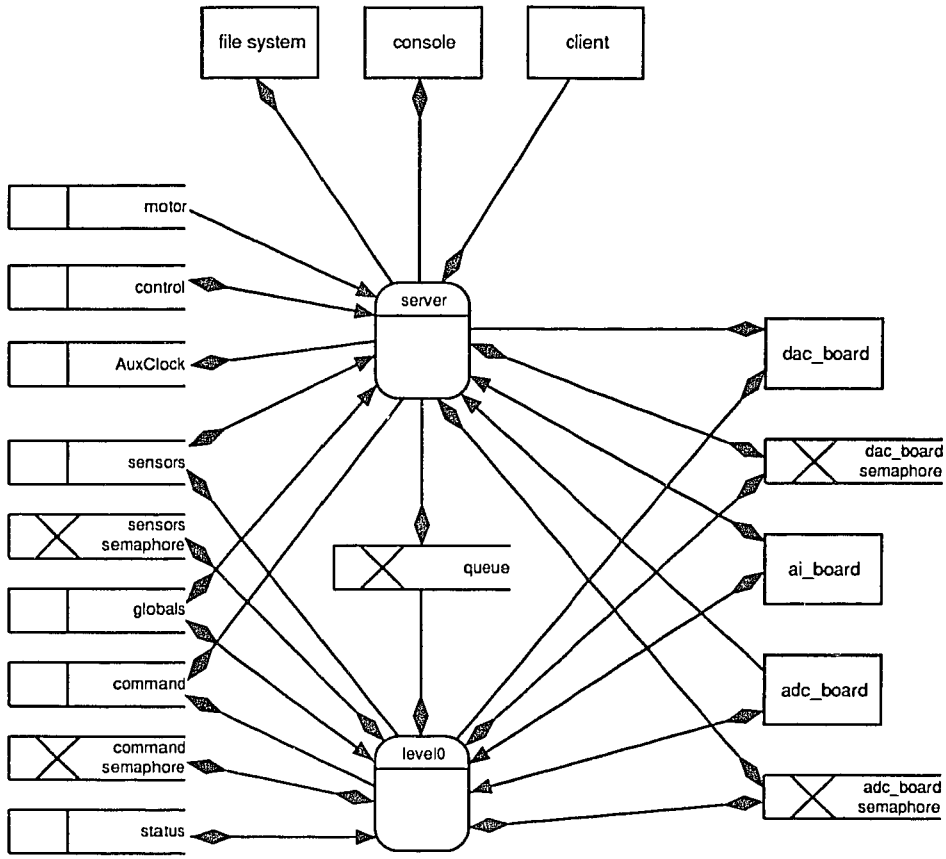


Figure 6.3: Dataflow diagram of robot control software

flows connecting two objects in opposite directions are combined into a single line with two arrowheads. The type of arrowhead at each end indicates the type of flow in that direction. The workload was specified using the graph file listed in Appendix B. All terminators were simulated. The structure of the functions was based directly on the structure of the actual server and level0 source code. All computation was replaced with synthetic operations. The control constructs were replaced with the loop and probabilistic branching constructs, as appropriate. None of the robot control code was used in the SW functions.

The SWSL function file for the server and the level0 control task is shown in Appendix D. The structure of these functions was based directly on the structure of the actual server and level0 source code. We maintained the control constructs and abstracted out the computation. All computation was replaced with synthetic operations. The control constructs were replaced with `LOOP` and `SWITCH` constructs, as appropriate.

This approach was taken because the goal of this experiment was to show that

Operation	Description
fp(iterations)	One of four floating point arithmetic operations per iteration
trig(iterations	One of six floating point trigonometric operations per iteration
bool(iterations)	Eight boolean operations (a combination of AND, OR, NOT, XOR, and logical shifts) per iteration
sread(INPUT label, wait flag)	Read one element from an INPUT; wait flag = WAIT indicates a blocking read; wait flag = NOWAIT indicates a nonblocking read
swrite(OUTPUT label)	Write one element to an OUTPUT

Table 6.1: Synthetic operations used in the robot control synthetic workload

the SW was capable of accurately modeling the robot control software. By representing the structure of the software exactly, we expected to achieve a representative SW. This approach is realistic if one considers the suggested operating environment for the SW. In Chapter 3, we stated that one goal of the SWG is to allow it to be incorporated into a CASE tool. In such a situation, both the dataflow representation of the workload and at least an outline of the control constructs and operations of each task should be available. It is exactly this information that we are using here. Therefore, we are making a fair and realistic comparison.

All computation and communication functions in the workload were abstracted and replaced by combinations of five operations. These operations are shown in Table 6.1. The computation operations were chosen because they best represented the types of computations that were performed in the workload. The calculation of the next joint positions in the adaptive control algorithm was floating point intensive. Therefore, we chose floating point operations for our SW. The `fp()` and `trig()` functions allow the user to specify exactly how many operations are to be executed, but the selection of the functions was evenly distributed between the four arithmetic operations in `fp()` and the six trigonometric and hyperbolic trigonometric functions in `trig()`.

The iteration values for the floating point operations were chosen based on the number of operations performed within each straight-line code segment. The percentages used in the `SWITCH` statements were based on the expected operating behavior of the actual workload. The workload was expected to execute primarily in the position interpolated mode. We were only concerned with steady-state performance of the system. We ignore the startup calibration stage of execution when the robot system was in standby mode.

Therefore, all SWITCH statements which represent checks for modes other than position interpolated had percentages of 0. Likewise, any code which dealt with other startup conditions had been eliminated.

LOOP constructs were used for repeated reads and writes to terminators. In addition, an infinite LOOP was used by the server function to make it a data-driven sporadic task. No loops were needed in the computation sections due to the iteration parameter for the synthetic operations.

6.2.4 Tuning the SW

If, after executing the SW, it was determined that it was not sufficiently representative of $\mathcal{W}$, the following steps were to be used to tune the SW.

1. Measure the overhead for calculating random numbers for the probabilistic branches. Compensate for this time by reducing the amount of computation within the blocks in each alternative path of the branch. The means of affecting this change is to reduce the iteration values for the `fp()` and `trig()` calls.

2. Adjust the branching probabilities to match observed operation modes.

6.3 Representativeness Experiments

6.3.1 Experimental design

The experiments followed a single factor design. The primary factor was the choice of workload, either the robotic software or the SW. The secondary factors, the system architecture and operating system, were held constant. Two response variables were measured. The first was the mean task execution time of the level0 task. This value determines the average time between when the joint position sensors were read and the next set of goal position values were produced. A short execution time was desirable. During execution, the adaptive control algorithm produces goal positions based on current position. This position is read from the joint position sensors at the beginning of the task's execution. The goal joint position is produced at the end of the execution. If the execution time of the level0 task is long, then the actual arm position at the end of the execution will be significantly different from the position measured at the beginning of the execution. Therefore, the goal positions will be based on out-of-date information. The result is jerky movement by the robot arm.

The second variable to be measured was the level0 task execution time distribution. The task execution time was determined by the control constructs within the task. The number of branches in the function were relatively few but significant. The distribution indicated the maximum and minimum execution times. It also indicated the effects of the level0 task execution on CPU utilization and the scheduling of other tasks.

Both of these indices relate to the level0 task. Neither measures the execution of the server task. The reason for this is that the level0 task is the only critical task in the workload. Its correct and timely execution determines the success of the application. The server is a relatively low priority task with no strict performance requirements other than correct execution.

A simple mechanism was used to collect the necessary performance data. Calls to a function called `probe()` were placed at the beginning and end of the level0 code. `probe()` calculated the elapsed time since the last call to `probe()`. For the call to probe at the end of the level0 task, this value was the task execution time. Each time `probe()` was called, it calculated a value $\Delta t_i = t_i - t_{i-1}$. The values $\sum_{i=0}^{n} \Delta t_i$ and $\sum_{i=0}^{n} \Delta t_i^2$ were calculated iteratively. These values are used to compute the sample mean and standard deviation. `probe()` also maintained a histogram of the Δt_i values. The histogram is used to calculate the task execution time distribution.

6.3.2 Experiments using the robot control software

Before running the experiments to measure the performance while executing the robot control software, we needed to determine the number of data samples that would be required. To measure a sample mean with a desired level of accuracy with a given confidence level, we require n samples, where n is computed as

$$n = \left(\frac{100zs}{r\overline{x}}\right)^2$$

where z is the normal variate at the desired confidence level, $\overline{x}$ is the sample mean, s is the sample standard deviation, and r is the percentage of accuracy desired [34]. In our preliminary measurements, we measured $\overline{x} = 8.7613$ and $s = 0.4817$. If we wanted 1% accuracy at a confidence level of 99%, then $r = 1$ and $z = 2.576$. We get

$$n = \left(\frac{100(2.576)(0.4817)}{(1)(8.7613)}\right)^2 = 200.59$$

Therefore, we had to collect at least 201 samples. Since the level0 task executed at 60 Hz and one sample was collected during each execution, we needed to collect data for at least 3.3 seconds to obtain the desired accuracy.

```
move(10.0,0.2,pt1,absolute);
wait();
put_gripper(open);
move(10.0,0.2,pt2,absolute);
put_gripper(closed);
move(10.0,0.2,pt3,absolute);
put_gripper(open);
move(10.0,0.2,pt4,absolute);
put_gripper(closed);
move(10.0,0.2,pt1,absolute);
put_gripper(open);
wait();
```

Table 6.2: Command script used to control the robot

Commands received from the client task control the actions performed by the robot. For our experiments, we used the commands listed in Table 6.2. This command script was taken from a demonstration script developed at the Robotics Laboratory. The commands are as follows.

- `move(t, acct, point, mode)` moves the robot arm to position `point` in `t` seconds. The `acct` parameter determines which portion of the movement time is to be used for each of acceleration and deceleration. `mode` may be either `relative` or `absolute`, indicating the coding used for `point`.
- `wait()` waits until the previous command completes before submitting another command.
- `put_gripper(open/closed)` opens or closes the gripper.

The command script moved the robot to five positions (labeled `pt1`, ..., `pt4`). At the end of the run, the arm had returned to its original position. At each position the gripper was opened or closed.

6.3.3 Experiments using the synthetic workload

The SW was run to closely approximate the duration and actions of the robot software. `SWITCH` percentages were chosen to reflect the distribution of command types and the execution duration of the `move()` commands. The same measurement mechanisms were used and the same statistics were calculated.

Time (msec)	Actual Workload	SW_1	SW_2
0	0	0	0
1	0	0	0
2	0	0	0
3	3	0	0
4	1	0	0
5	0	0	0
6	0	0	0
7	0	0	4
8	4216	1	4246
9	732	910	661
10	62	2256	89
11	0	1833	0
12	0	0	0
13	0	0	0
14	0	0	0
15	0	0	0
16	0	0	0
17	0	0	0

Table 6.3: Task execution time histogram.

6.4 Experimental Results

We first executed the robot control software. We then executed the SW and compared the results. Our first SW, SW_1, did not pass the tests described below. Its task execution time histogram is included in Table 6.3. By looking at the histogram, it was obvious that the mean was too high and the distribution was skewed to the high end. We then tuned the SW as indicated in Section 6.2.4. To lower the mean, we scaled the number of floating point operations to 70% of their original number. This scaling is shown in Appendix D by the use of the `fp_scaling_factor` constant in the `fp()` synthetic operations. To change the distribution we reexamined the `SWITCH` probabilities in the function file. We found a block with an incorrect percentage value corresponding to the probability of the robot operating in a certain mode[2]. We changed the percentage to a more appropriate value and reran the experiment. This second SW, SW_2, produced the performance values used in the analysis below.

[2]This percentage is noted in the listing in Appendix D.

Time (msec)	$F_{n_1}(x)$	$F_{n_2,SW_1}(x)$	$\lvert F_{n_1}(x) - F_{n_2,SW_1}(x)\rvert$
0	0.0	0.0	0.0
1	0.0	0.0	0.0
2	0.0	0.0	0.0
3	0.000598	0.0	0.000598
4	0.000798	0.0	0.000798
5	0.000798	0.0	0.000798
6	0.000798	0.0	0.000798
7	0.000798	0.0	0.000798
8	0.84164	0.0002	0.84144
9	0.98763	0.1822	0.8054
10	1.0	0.6334	0.36661
11	1.0	1.0	0.0
12	1.0	1.0	0.0

Table 6.4: Task execution time cumulative distribution functions.

Time (msec)	$F_{n_1}(x)$	$F_{n_2,SW_2}(x)$	$\lvert F_{n_1}(x) - F_{n_2,SW_2}(x)\rvert$
0	0.0	0.0	0.0
1	0.0	0.0	0.0
2	0.0	0.0	0.0
3	0.000598	0.0	0.000598
4	0.000798	0.0	0.000798
5	0.000798	0.0	0.000798
6	0.000798	0.0	0.000798
7	0.000798	0.0008	0.00000223
8	0.84164	0.85	0.00835660
9	0.98763	0.9822	0.00543462
10	1.0	1.0	0.0
11	1.0	1.0	0.0
12	1.0	1.0	0.0

Table 6.5: Task execution time cumulative distribution functions.

6.4.1 Comparison of mean execution time

Our first test is to compare the means of the distributions for samples denoted 1 and 2. Sample 1 is the data from the robot control software, sample 2 is from the SW. The clock by which the execution times was measured had a granularity of 1 millisecond. We collected approximately 5000 data points for each workload. Therefore, the sample means are accurate to 0.015 milliseconds with 99% confidence. This value corresponds to an accuracy of 0.18%. The sample means for both workloads equaled 8.17 milliseconds. The sample standard deviations were 0.43 for sample 1 and 0.42 for sample 2. Since the means and standard deviations were identical at this level of accuracy, we conclude that the means of the distributions are equal at the 99% confidence level.

6.4.2 Comparison of the task execution time distributions

To compare the distributions of the execution time we use a Kolmogorov-Smirnov two-sample test as described in [5]. This test is used to compare two cumulative distribution functions to determine if they are based on samples drawn from the same population. To use this test, we must transform the histograms of the sampled data into cumulative distribution functions $F_{n_i}(x)$ such that

$$F_{n_i}(x) = \frac{k}{n_i}$$

where n_i is the number of samples in distribution i, and k is the number of observations less than or equal to x. For a two-tailed test, we calculate

$$D_{n_1,n_2} = \max |F_{n_1}(x) - F_{n_2}(x)|$$

and reject the null hypothesis if D_{n_1,n_2} exceeds the critical value. For a confidence level of 99%, the critical value is $1.63\sqrt{\frac{n_1+n_2}{n_1 n_2}}$.

The data collected in our experiments are shown in Table 6.3. We collected $n_1 = 5014$ data samples for the robot control software and $n_2 = 5000$ samples for the SW. Therefore, the critical value for the Kolmogorov-Smirnov test is

$$1.63\sqrt{\frac{5000 + 5014}{5000 \cdot 5014}} = 0.0325.$$

The cumulative distribution functions and the D_{n_1,n_2} values are shown in Tables 6.4 and 6.5. $F_{n_1}(x)$ is the cumulative distribution for the task execution times for the robot control software. $F_{n_2,SW_1}(x)$ and $F_{n_2,SW_2}(x)$ are the cumulative distribution functions for SW_1 and SW_2, respectively. The maximum absolute value of the difference, $|F_{n_1}(x) - F_{n_2,SW_2}(x)|$, is 0.008357. This value is not greater than the critical value. Therefore, with a 99%

confidence level, we cannot reject the hypothesis that the distribution of the execution time of the SW task is identical to the execution time of the actual task. Thus, we establish the representativeness of the SW.

6.5 Summary and Discussion

In this chapter, we described experiments performed to demonstrate the ability of the SWG to produce representative SWs. We used the SWG to produce an SW which modeled a robotics application. We used a performance based model of representativeness to compare the SW with the actual application software. The results showed that the SW was able to elicit the same performance from the system as the actual workload with a high degree of accuracy.

The primary result of this evaluation is that the SWG was able to produce a representative SW. In fact, once the initial structure and parameters for the SW were chosen, only one revision was required to obtain representativeness with the desired accuracy. We can therefore state, with some degree of confidence, that the structures and parameters that were designed into SWSL are sufficient for specifying this type of workload.

The evaluation also allows us to make some observations about the use of the SWG. The first of these observations is that minimal porting effort was required to get the SWG to produce a running SW for the robot control computer. The SW was originally implemented on HARTS. We only needed to change the system-dependent system calls in the driver and synthetic operations in the library of operations. Some minor changes were required to the SWG. However, these were only required to support multiple target operating systems. We added the special constant `OPERATING_SYSTEM` to be used in the SWSL specification. The use of this constant is shown in Appendix E. Its value is used by the SWG to indicate the target operating system. The SWG produces the appropriate code for that target. These changes to the SWG would not need to be repeated to port the SWG to yet another operating system.

The time required to port the SW and SWG was approximately two man-weeks. Before the porting effort, we had no previous experience with VxWorks. So the porting time includes the lead time required to become familiar with VxWorks and its programming environment. A user who is experienced with the target system should be able to port the SWG to their target system in less than a week.

We also observed that it was relatively easy to produce a representative SW if one has the source code of the workload to use as a guide for writing the SWSL specifica-

tion. In most cases, however, the users of the SWG will not have the source code of the workload available. Instead, they will have to specify the SW using other means. In much the same way as a rapid prototype is developed, the structure of the SW can be derived from the high-level requirements specifications or structured analysis specifications for the workload. These specifications will provide information which is equivalent to the specification of the transformations, stores, terminators, and flows in our workload model. The system-dependent parameters may be derived from these high-level specifications, or they may be estimated based on the user's experience with other real-time workloads. Lower level structural details may be filled in as their design becomes available, or they may be estimated. For example, if the internal structure of a task is not known, its computation may be modeled using a loop that executes a random number of times with a specified distribution. This model of a task may be further refined by the user if an even lower level of detail is needed. Another technique would be to use SWSL functions from other SWs to represent functions with similar characteristics in the SW being defined. These functions would also be estimates of the actual behavior of the workload's tasks, but the user would be more familiar with them and thus more aware of their level of representativeness.

These techniques will allow the user to produce an approximately representative SW. By the definition of representativeness, the actual level of representativeness can only be measured if the real workload is available for comparison to the SW. However, even if the real workload is not available, the level of representativeness can be estimated by comparing the performance of the SW with the performance of the modeled workload on other target systems and taking into account the differences in the systems. Alternatively, the representativeness can be estimated by comparing the performance of the SW with the performance of workloads which are similar to the real workload and are run on the same target system as the SW.

As stated in Chapter 2, increased representativeness requires more information about the workload being modeled and more space to store this information. An example of this fact can be seen by comparing the function specification in Appendix D with the function specifications[3] in Appendix I. The functions in Appendix D were designed to reproduce the computation and system call execution behavior of the workload, and are therefore much more complex and lengthy. The functions require information about the loop counts, branching probabilities, number and type of system calls, and amount and type of computation. This level of detail is necessary if the function is to reproduce the

[3]These functions are described in Chapter 7

number, sequence, and frequency of computation instructions and system calls of a task in an actual workload. In contrast, the functions in Appendix I were designed to use a single resource without representing the low-level behavior of a workload. Thus, the specification is simple and compact.

The representativeness will also be affected by the accuracy of the workload characterization information. For example, inaccurate estimations of the loop count of an outer loop or inaccurate branching probabilities in a major branch can cause vastly different behaviors by the functions. Thus, if a high level of representativeness is required, the user must obtain a large amount of accurate data about the workload and accurately model it in the SWSL specification. Of course, the requirement for a large amount of very accurate data is only necessary if the SW is to be representative of a specific workload. If the user only requires an SW which is representative of a class of real-time workloads, then less information is needed, and the more approximate construction techniques will be sufficient.

CHAPTER 7

USING THE SYNTHETIC WORKLOAD FOR PERFORMANCE EVALUATION

7.1 Introduction

In this chapter we will demonstrate the other capabilities of the SWG by using it to evaluate the performance of an experimental distributed real-time system. The SWG is capable of producing representative SWs. As such, it may be used in evaluations aimed at determining the performance of a system under specific, expected workload conditions. However, such evaluations are not the sole purpose of the SWG. It is also capable of supporting controlled experimental evaluation of systems under a range of workload conditions. The SW used in such an evaluation does not necessarily represent any actual workload. Instead, the SW represents specific combinations of workload characteristics whose effects on performance are being characterized. We demonstrate this latter capability in this chapter.

We have designed a series of experiments which were performed on a distributed real-time system. In this chapter, we will identify the workload characteristics which were required to perform the experiments. An SW was constructed which exhibited these characteristics. We will discuss in detail the specification of the SW using SWSL. The specification will be related to the requirements of the experimental design. Then, the results of the experiments will be analyzed and conclusions drawn.

In these experiments, we measured the performance of the communication software for the distributed real-time system HARTS (Hexagonal Architecture for Real-Time Systems). HARTS is being developed at the Real-Time Computing Laboratory of the University of Michigan as an experimental testbed for exploring hardware and software techniques for real-time communication in a distributed multiprocessor. The experiments determine which workload characteristics most affect the performance of the system. The characteristics under study were varied in a controlled manner while all other characteristics were fixed at specific levels.

This chapter is organized as follows. Section 7.2 describes the target system for this evaluation. Section 7.3 describes experimental design used in this evaluation. The specification of the synthetic workload is presented in Section 7.4. The experimental results are presented in Section 7.5. In Section 7.6, we present our conclusions.

7.2 Target System

The target system for this evaluation was HARTS. In the following subsections, we describe the relevant details of HARTS and its operating system HARTOS. We also describe HMON, a monitor for real-time systems, which was used to collect the performance data used in this evaluation.

7.2.1 HARTS

We are currently constructing a 19-node version of HARTS. Each HARTS node is a shared memory multiprocessor formed by up to three Motorola 68020 microprocessors which serve as the application processors (APs) for the node. The architecture of a node and the operating environment are shown in Fig. 7.1. In the final version, the multiprocessor nodes will be interconnected via a wrapped hexagonal mesh interconnection network [12, 17]. A hexagonal mesh is a 6-regular homogeneous graph. The 19-node hexagonal mesh is shown in Fig. 7.2. The arrows at the edges indicate links that are "wrapped" from one edge node to another. The APs are to be connected to the network by custom-designed communication hardware, called the network processor (NP). The NP executes the bulk of the communication protocol software, thus relieving the APs of this task. All nodes are connected to a workstation by a shared Ethernet. The workstation is also connected to the campus computing facilities by a separate Ethernet connection. In this way, programs developed and compiled on other commonly used workstations may be downloaded to HARTS, but HARTS executes with a dedicated local Ethernet. The workstation also serves as the console for the HARTS nodes. It is connected via a multiplexor (MUX) to the console serial ports on the System Controller cards of each HARTS node. These serial connections are primarily used for remote debugging. While the hexagonal mesh network is being designed and built, the HARTOS software executes on the Ethernet processor (ENP) and uses the Ethernet as the communication medium. Most of the operating system and software developed with the Ethernet is expected to be portable to the hexagonal mesh interconnection network.

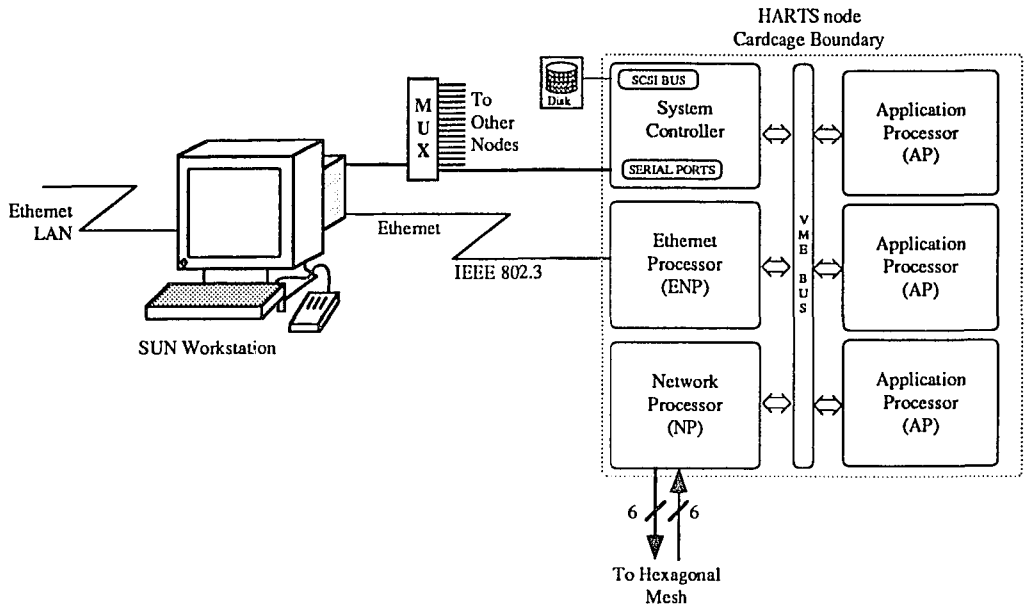


Figure 7.1: HARTS node architecture and operating environment.

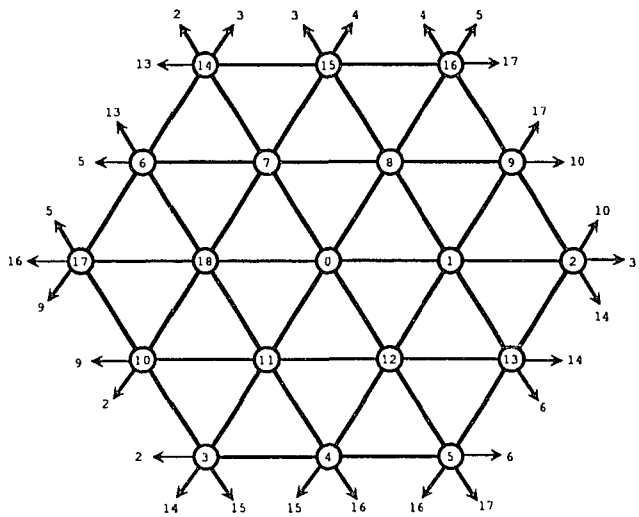


Figure 7.2: Wrapped hexagonal mesh with nineteen nodes.

7.2.2 HARTOS

A preliminary version of HARTOS has been completed and is operational [38]. It is built upon the real-time kernel pSOS [61] which provides a number of basic real-time kernel functions. A complete discussion of pSOS may be found in [61]. We present here a brief description of pSOS and HARTOS as they apply to this evaluation.

pSOS

pSOS uses a process model of computation. Scheduling of processes is priority-based with preemptable and nonpreemptable modes. Communication between processes is via event signalling and message passing through structures called message exchanges. Each process has fifteen events associated with it [61]. A process may signal one or more events to another process. A process may wait for one or all of several events. Events contain no data and are not queued; if an event is pending, then a subsequent signalling of that event will be ignored. Exchanges are data structures where short, fixed-length messages and processes may be queued. If a process is queued at an exchange and a message arrives, the message is copied to the process's buffer area and the process is dequeued and made ready to execute. Likewise, if a message is queued at the exchange and a process performs a system call to receive a message, the message is copied to the process. The process continues execution and the message is dequeued.

HARTOS Services

Version 1 of HARTOS was developed to extend the pSOS interprocess communication mechanisms to operate in a distributed multiprocessor environment. To support these facilities, HARTOS provides a name service whereby processes may locate remote objects with which they must interact. It does not provide support for time-constrained communication. Version 2 of HARTOS, which is currently under development, is designed to provide such support. At this time, version 2 of HARTOS is not yet complete. Therefore, version 1 was used for this evaluation.

The programmer's interface to HARTOS is via a number of parameterized system calls. A user task has control over several parameters in their execution, including the number of retries and whether the call is blocking or non-blocking. A blocking call is one in which the user task waits until the operation is completed on the remote processor and the results are returned. For a blocking call, a task can specify a timeout period which is the maximum duration that it is willing to wait. For a nonblocking call, the task continues

execution immediately upon placing the call. The NP handles the call processing but no results are returned to the calling task.

The distributed kernel functions are handled using a variation of the remote procedure call (RPC) mechanism [6]. At the logical level, the operations involved in a remote function call can be described as follows. The processor AP1 forwards the data to the NP. The NP marshals the data into a packet and transmits the packet to the NP (NP2) on the destination node. NP2 now interprets the message, determines the destination AP and forwards the message to that AP (say AP2). AP2 executes the function and returns the reply to NP2. The reply then traces its way back to AP1. The server side handling for these functions is non-blocking and very short and is executed as part of the interrupt handler. This eliminates the need for kernel server processes which are used in many RPC systems.

HARTOS Implementation

The communication system for HARTOS consists of agents on both the AP and the NP. On the AP side there are reentrant stub interface routines for the calls, a common network agent to communicate with the NP, and an interrupt handler. Communication between the APs and the NP is through mailboxes in the memory of each AP. Two mailboxes are used on each AP. One is for passing data to the NP for a remote call. The reentrant stub interface routines extract the call parameters and place them into a mailbox. They then trap to the network agent which synchronizes access to the NP and places the request into the mailbox. If the function specifies a blocking operation mode, the process is then suspended awaiting a completion signal from the NP. Once the parameters of the call are placed in the mailbox, a flag is set in the memory of the NP. Access conflicts between processes on the AP are prevented by using a non-preempt mode while accessing the mailbox. The other mailbox is used to pass data from the NP to the AP. This data can be either the parameters of a remotely generated request or a reply from a call which originated from the AP. When data is placed in this mailbox, the interrupt handler is invoked to execute the remote call or to transfer the results to the AP. Access to the mailboxes is governed by a simple producer-consumer type protocol with a buffer size of one.

The NP program consists of several logical processes corresponding to handlers for different operations. There are separate handlers for message send, packet receive, and timeout. These are scheduled by a control loop which polls the flags associated with the AP mailboxes and such structures as the queue of incoming packets. When one of these data structures indicates that a task is to be performed, the appropriate handler is activated. The handlers run to completion and there is no processing done during interrupts.

The interface between the NP program and the Ethernet Controller (LANCE) is through the K1 kernel [14]. The K1 kernel controls the sending and receiving of messages. Outgoing messages are submitted to the K1 kernel by the NP program. The K1 kernel queues the messages for transmission by the LANCE. It also provides an interrupt service routine (ISR) to pass received packets to the NP program. When a packet arrives for the NP, the K1 kernel receive ISR calls the NP program's receive ISR which places the packet on a queue of newly received packets. The status of this queue is checked during the NP program polling sequence. When a packet is present on the queue, the receive handler is called.

The K1 kernel also provides a timeout handling facility which is used by the NP program. The NP program sets timeouts for outgoing messages and transactions if requested by the application process. When a timeout occurs, the K1 kernel activates the timer ISR which queues the timeout control data structure on the timeout queue. This queue is also checked during the NP program polling sequence and the timeout handler is activated if a timeout has occurred.

7.2.3 HMON

To collect execution data during the evaluation, we use HMON, a monitor for distributed real-time systems [16]. HMON was designed to support services like debugging distributed real-time applications, aiding real-time task scheduling, and measuring performance. Monitoring is performed transparently so the programmer is not forced to add special monitoring code to applications. HMON is flexible enough to observe both high-level events that are operating system- and application- specific as well as low-level events like shared variable references.

The HMON runs on a dedicated AP, called the monitor processor (MP), on each node of HARTS. Additional code to collect data runs on the NP and the APs of each node (see Fig. 7.3). The monitoring system can be divided into three phases: data extraction on the APs and NP, data compression on the monitoring processor, and uploading logged data to an external workstation. Data on monitored events is acquired through code inserted into the monitored software. System calls are monitored transparently by using pSOS and HARTOS system call interfaces which have been instrumented to produce monitor data. HMON also monitors interrupts and context switches. All extracted data is sent to the MP where it is ordered and compressed by a process executing on the MP. At the end of an experiment, data is uploaded from the MP to a workstation outside HARTS.

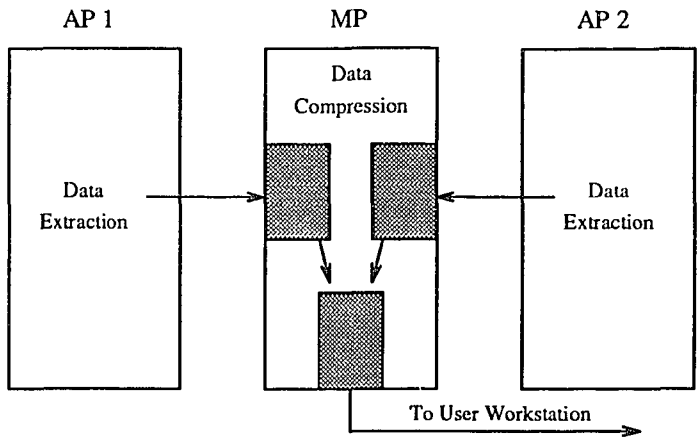


Figure 7.3: Monitor data collection.

7.3 Experimental Design

Simple, baseline measurements of HARTOS performance were reported in [38]. That evaluation was limited to communication between two nodes, each with a single process executing on a single processor. By using the SWG, we can perform much more sophisticated evaluations. The evaluation described in this section was aimed at determining which workload and system configurations most affected the performance of HARTOS.

We wanted to compare the effects of various factors on the performance of the HARTOS communication facilities. We used a $2^k r$ full factorial experiment with two levels for each of k factors and r repetitions of each experiment. Four primary factors were used. They were: number of nodes (N), number of processors per node (P), number of sending tasks per processor (T), and number of sequential send calls per task (C). Each experiment was repeated $r = 5$ times to allow us to estimate the experimental error. Factor combinations for each experiment will be indicated by the tuple (N, P, T, C).

To allow comparison of effects, each factor was evaluated at two levels, one with a low value and one with a high value. The low values were determined by logical limitations. All factors, except N, had low values of 1. The low value for N was 2 since at least two nodes are required to communicate over the network. The high value for N was 4. At the time of the evaluations, only 4 HARTS nodes were configured with enough APs for the experiments. The high value for P was 2. There can be up to three processors per node, but one is required for the monitor. Therefore, only two were available for the workload. The high values for T and C were chosen to be 10. Preliminary evaluations indicated that performance might level off with greater than 5 tasks. Therefore, it was decided that 10

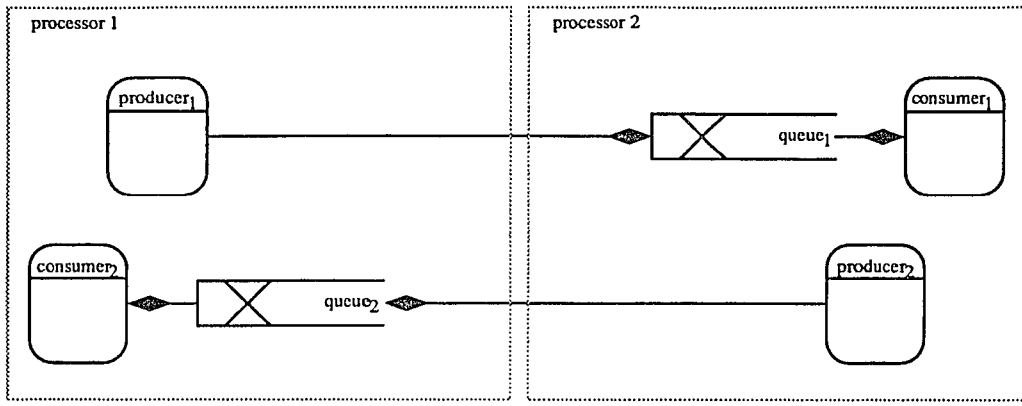


Figure 7.4: Producer-Consumer model for workload.

tasks would be sufficient to observe a maximum performance value.

Only the values of the primary factors were changed between experiments. A number of parameters of the workload were fixed. The only call type considered was the `rsend_x()` call. This call sends a short, fixed-length message to a remote message exchange. This call was chosen arbitrarily. It was shown in [38] that all remote calls had approximately the same execution times. Therefore, any call could have been used. All calls were blocking, and a fixed timeout was set for each call. No retries were allowed.

For each combination of factor levels, the workload was executed and two performance indices were measured. The first was the time, t, required to complete a blocking message send across the network. This value included the time to send the message and to receive the reply indicating that the message was delivered successfully. We will refer to this value as the message response time. The second performance index was the percentage of messages that were lost, m. This index was measured by counting the number of send requests that timed out. The count was verified by comparing the number of messages sent with the number of times that the receiving tasks received a message.

7.4 Synthetic Workload Specification

The SW used in Chapter 6 demonstrated the features of the SW that can be used when evaluating processor-level performance. In this chapter, we are interested in network performance. We are only interested in processor-level details insomuch as they influence network performance. Therefore, for this SW, the functions executed by the tasks were relatively simple in structure. The operations consisted primarily of synthetic operations to produce network traffic. We concentrated on the structure of the task graph and the

placement of communicating tasks such that network traffic was carefully controlled.

The workload for this experiment consisted of pairs of tasks with a simple producer-consumer communication relationship. Messages were sent from the producer to the exchange that was read by the consumer. An example of two pairs on two nodes is shown in Figure 7.4. This configuration corresponds to the $(2, 1, 1, C)$ workloads, for all values of C. For a given producer-consumer pair, the producer and consumer were always on separate nodes. There were T producers executing on each of the $N \times P$ active processors in each experiment. For workloads containing more than one pair, the producer, consumer, and exchange were specified using object templates and multiple instances of the objects were produced. The producer executed periodically. Periodic execution allowed us to obtain multiple measurements of t for each run of the experiment. These values were used to calculate the mean value for t for each experiment with a precision greater than the clock resolution would allow for individual measurements. During each period the producer executed a sequence of C remote system calls. Deadlines were defined to be slightly less than the period length. The difference between the deadline and the period was to give the SW time to process any missed deadlines. However, no deadlines should be missed because the period and deadline were defined to give sufficient time for all calls by all tasks on the processor to time out. Task priorities were fixed such that the sending tasks had higher priorities than the receiving tasks. We wanted the workload to produce messages at the maximum rate. Therefore, sending messages was more important. Since messages were queued at the receiving end, removing the messages from the exchanges was less critical. Although the exchange message queues had unlimited capacity, queueing messages used limited memory resources. Therefore, it was necessary that the consumer task be executed whenever the CPU was otherwise idle and there were queued messages. However, this process could be relegated to a lower priority level.

The graph file for the SW is listed in Appendix H. The graph file shown is actually the file used for the $(4, 1, 10, 10)$ workload. For organizational purposes we constructed separate graph files for each workload configuration. The files were identical except for the values of the parameters that determine the four experimental factors.

For this experiment, we have used a number of SWSL features. For example, we took advantage of the ability to define constants. Two constants, `num_tasks` and `loopcount`, are of primary importance. They defined the number of tasks and number of calls per task, respectively. The value for C was implemented directly through the `loopcount` constant. Its use in the functions file is discussed below. The `num_tasks` constant did not directly implement the value for T. Instead, this constant was used as the basis for calculating other

parameter values that depended on T. The number of tasks was implemented through the ACTIVE parameter for the tasks. This parameter indicated whether a task would execute during a given run of the SW. For the workloads with $T = 1$, the ACTIVE parameter for task1 and task2 was set to true. All other tasks had the parameter set to false. For the workloads with $T = 10$, all tasks had an ACTIVE parameter value of true. Other important parameters, such as the task period, task deadline, and the duration of the experiment, were defined as functions of these constants. With this technique, we only needed to change the values of these two constants and all parameters that depended on them were changed automatically.

The workload tasks and stores were defined in the DEFINITIONS section. For workloads with ten tasks, ten separate source and destination pairs were defined. This was necessary because objects on the same processor must be defined uniquely. Except for the values of the ACTIVE and PROCESSOR parameters, the object definitions did not vary between configurations. All other parameters that vary were replaced with constants.

The factors N and P were implemented through the values for the PROCESSOR parameters of the objects. Constants were used to define these values. We found it much easier to change the values when they were grouped together in the CONSTANTS section. For all of the $N = 2$ workloads, the producer tasks sent messages to consumer tasks located on the corresponding processor of the opposite node. For the $N = 4, T = 1$ workloads, the producers sent messages to the next node in sequence, modulo N. That is, processor 6 sent to processor 8; processor 8 sent to processor 9; processor 9 sent to processor 10; and processor 10 sent to processor 6[1]. The goal was to produce a more distributed traffic pattern. The $N = 4, T = 10$ workloads had a more complex processor assignment. The task graph corresponding to a portion of the processor assignment for the $(4, 1, 10, 10)$ workload is shown in Figure 7.5. The producer tasks, task1*, on node rtcl6, processor 1 (denoted rtcl61 in the graph file) are shown with their respective destination exchanges, store1*, and consumer tasks, task2*. The producer tasks for processors rtcl81, rtcl91, and rtcla1 had exactly the same relationship to their destination exchanges and tasks. For each set of producer tasks in these workloads, we distributed the destination exchanges and tasks among the three remaining nodes. The purpose was to distribute the communication load across the system. This version of HARTOS does not support distributed clock synchronization. Therefore, it was known before the experiments that the method used to synchronize the root tasks in the SW would provide loose synchronization at best. It was hypothesized that this loose

[1]The HARTS nodes used in these experiments were designated rtcl6, rtcl8, rtcl9, and rtclA.

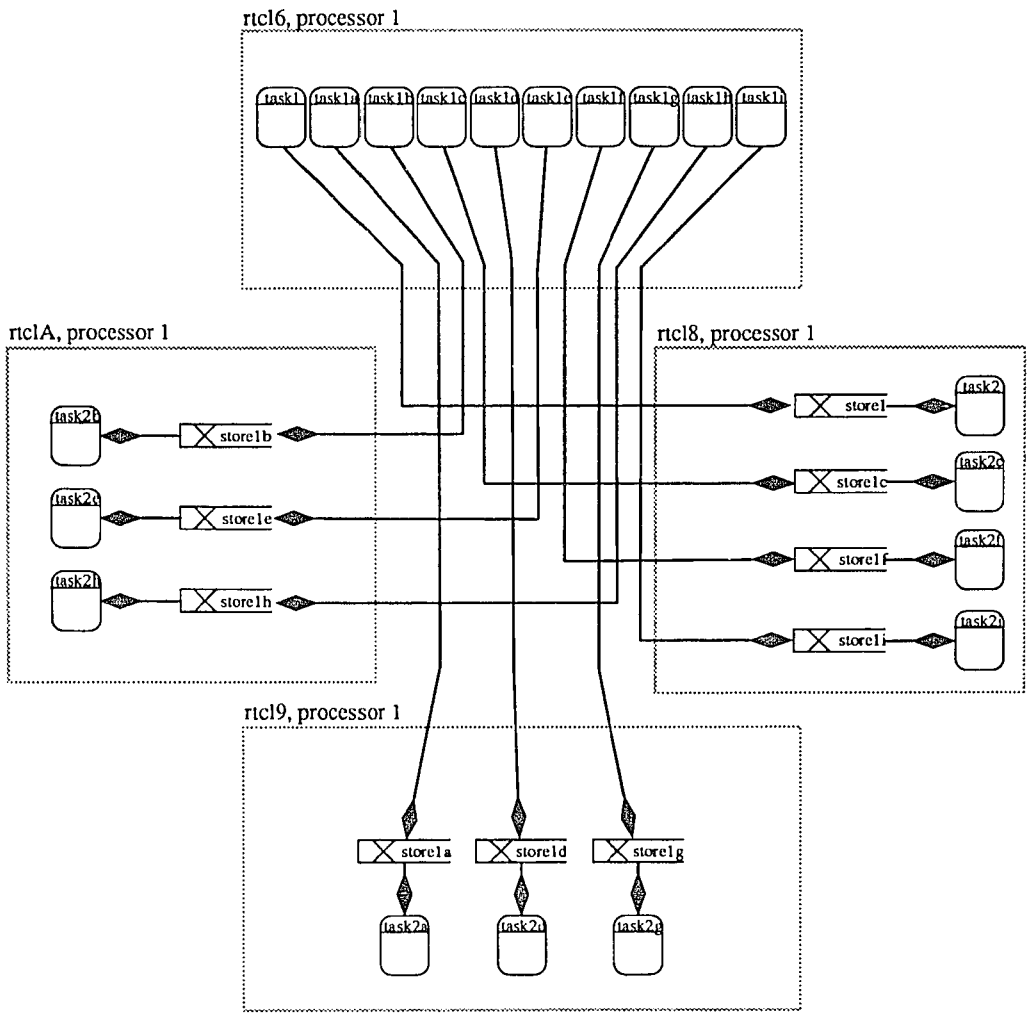


Figure 7.5: Workload (4, 1, 10, 10). Producer tasks on RTCL6 with corresponding consumer tasks.

synchronization might result in a relative misalignment of the periods of the producer tasks on different processors. As a result, there might be high contention for network resources between relatively synchronized processors, but little or no contention encountered by the less synchronized processors. Under these circumstances, the variance between the mean performance values measured on the different processors could be high. In an attempt to counteract this possibility, we distributed the load caused by a single node across the remaining nodes. The result was expected to be a higher probability that a given node would encounter contention from other nodes, and thus a reduction of the interprocessor variance. The actual results of this distribution of load is discussed in Section 7.5.

Two functions were used in this workload, `fun1()` and `fun2()`. `fun1()` was the producer and `fun2()` was the consumer. These functions are shown in Appendix I. These exact same functions specifications were used for all experiments. The value of `loopcount` was changed depending on the value of C for the workload. Since `loopcount` was defined in the graph file, no changes were made to the functions file for the different workload configurations.

The function structures were simple. During each invocation, `fun1()` sent messages to the exchange that was read by `fun2()`. The `loopcount` constant determined the number of messages sent. Messages were sent using the `swrite()` synthetic operation. The syntax for this operation is `swrite(output_name, wait_flag, retries, timeout)`. Because of the specification of the stores, the `swrite()` operation performed the remote send using the HARTOS `rsend_x()` call. Each call was given a timeout of 200 milliseconds. From preliminary measurements we determined that this was at least twice the expected maximum value for a given call. Therefore, unless the message was lost, each call should have completed before the timeout. No retries were specified, because we wanted to measure the time for a single attempt at each call. Retries could introduce a multimodal distribution for the message response time; we wanted to avoid this possibility. `fun2()` executed a blocking read command within an infinite loop. It read as many messages as were received by the exchange and blocked on the read call if no messages were present.

The experiment file defines the experimental parameters. Due to the experimental instrumentation used for this evaluation, we could not use the multiple run facility. For some unexplained reason, the process of uploading the HMON data after an execution of the SW caused subsequent runs of the SW to fail. Hence, we were required to reset the HARTS hardware and download the software between runs. Therefore, the experiments were defined to have a single run. All processors were to have the same experimental parameters. Therefore, the `default` processor label was used. The length of each run was

based on the loop count, number of tasks, and period of the task. The goal was to run the experiment until a total of approximately 1000 messages had been sent by the tasks on each processor.

7.5 Experimental Results

For each of the sixteen factor combinations, five experiments were run and measurements were made. During the experiments, the SW was the only workload executing on HARTS. The monitor collected data on all pSOS and HARTOS calls. It also recorded the occurrence of all interrupts. Thus, we were able to count time by noting the timing interrupts. Using this information, we recorded the time that each `rsend_x()` call was made and the time that the reply was received, i.e., the reply mailbox interrupt occurred. By recording the time that the reply interrupt occurred instead of recording the time that the task received the message, we eliminated from the response time the delay caused by multiprocessing on the processor. The response time only included the message processing time; it did not include the scheduling delay incurred by the task. The shortest interval for the clock interrupts allowed by the system was 1 millisecond. Therefore, our timing granularity was restricted to this value.

To analyze the data, we use a multiplicative model for the predicted value of the performance index as a function of the factors. We chose a multiplicative model based on the expectation that the performance of the system is a function of total network load, and the observation that the total network load is determined by the product of the various factors. For example, when we increased T from 1 to 10, we increased the number of tasks on *each* processor. Therefore, the total load on the network was multiplied by 10, e.g., the $(4, 2, 1, 1)$ workload contained 8 tasks, but the $(4, 2, 10, 1)$ workload contained 80 tasks.

The model we use is

$$y_{ij} = e^{q_0} e^{q_N x_N} e^{q_P x_P} e^{q_T x_T} e^{q_C x_C} e^{q_{NP} x_N x_P} \ldots e^{q_{NPTC} x_N x_P x_T x_C} e^{E_{ij}}. \tag{7.1}$$

Where y_{ij} is the value of the performance index for the jth experiment using the ith workload configuration, the qs are the effects of the various factors and combinations of

Factor levels T, N, C, P	$\bar{t}$	s_t	$\overline{\ln t}$
1, 2, 1, 1	5.67	0.376	1.73
1, 2, 1, 2	5.0	0.00019	1.61
1, 2, 10, 1	5.05	0.359	1.62
1, 2, 10, 2	5.13	0.0128	1.63
1, 4, 1, 1	4.64	0.0662	1.53
1, 4, 1, 2	4.75	0.00315	1.56
1, 4, 10, 1	5.22	0.0479	1.65
1, 4, 10, 2	5.05	0.109	1.62
10, 2, 1, 1	18.8	4.8	2.90
10, 2, 1, 2	18.3	6.86	2.84
10, 2, 10, 1	26.4	4.82	3.26
10, 2, 10, 2	36.4	8.04	3.57
10, 4, 1, 1	15.6	5.13	2.70
10, 4, 1, 2	10.6	0.678	2.36
10, 4, 10, 1	11.7	1.24	2.45
10, 4, 10, 2	12.5	2.0	2.52

Table 7.1: Results for response time: mean $\bar{t}$, standard deviation s_t, and mean of the transformed data $\overline{\ln t}$.

factors, E_{ij} is the error, and

$$x_N = \begin{cases} -1 & \text{if 2 nodes} \\ 1 & \text{if 4 nodes} \end{cases}$$

$$x_P = \begin{cases} -1 & \text{if 1 processor per node} \\ 1 & \text{if 2 processors per node} \end{cases}$$

$$x_T = \begin{cases} -1 & \text{if 1 sending task per processor} \\ 1 & \text{if 10 sending tasks per processor} \end{cases}$$

$$x_C = \begin{cases} -1 & \text{if 1 call per task} \\ 1 & \text{if 10 calls per task.} \end{cases}$$

If we perform a log transform on the input data, we get a nonlinear regression model of the form

$$\ln(y_{ij}) = q_0 + q_N x_N + q_P x_P + q_T x_T + q_C x_C + q_{NP} x_N x_P + \ldots + q_{NPTC} x_N x_P x_T x_C + E_{ij}. \quad (7.2)$$

We can easily solve this model for the qs using known techniques for nonlinear regression models [34].

Table 7.1 shows the results of the experiments for the response time, t. The reason for the reordering of the factors will become apparent later. For each factor level

Effect	Value	90% Confidence Interval
q_0	2.22	[2.19, 2.26]
q_N	-0.1748	[-0.208, -0.139]
q_P	-0.00850	[-0.0431, 0.0261]
q_T	0.602	[0.568, 0.637]
q_C	0.0684	[0.0338, 0.103]
q_{NP}	-0.0274	[-0.0621, 0.00717]
q_{NT}	-0.145	[-0.18, -0.11]
q_{NC}	-0.0566	[-0.0912, -0.0219]
q_{PT}	0.00571	[-0.0289, 0.0403]
q_{PC}	0.0537	[0.0191, 0.0883]
q_{TC}	0.0574	[0.0228, 0.0921]
q_{NPT}	-0.0394	[-0.074, -0.00479]
q_{NPC}	-0.0104	[-0.045, 0.0243]
q_{NTC}	-0.0903	[-0.125, -0.0556]
q_{PTC}	0.0431	[0.00844, 0.0777]
q_{NPTC}	0.0144	[-0.0202, 0.049]

Table 7.2: Effects for the multiplicative model for t.

combination, we see the mean of the response time, $\overline{t}$, the standard deviation, s_t, and the mean of the log of the response time, $\overline{\ln t}$. The values for the response time do not include the response time (actually the timeout time) for calls where the messages were lost. The untransformed values, $\overline{t}$ and s_t, are presented for reference only. All analysis was done with the transformed values.

Solving for the q values is done using the sign table technique described in [34]. This method constructs the 2^k linear equations for $\hat{y}_i$, the predicted value of y_i. It then solves the system of equations for the q values. Using this technique and solving for the q values, we get the results in Table 7.2. The 90% confidence intervals for the effects are also given. If the confidence interval contains 0, then the effect is not significant at this confidence level. Plugging the significant values into Equation 7.2, we get

Factor(s)	Percentage
T	77.5
N	6.45
ERROR	5.87
NT	4.49
NTC	1.74
C	1.0
TC	0.704
NC	0.683
PC	0.616
PTC	0.396
NPT	0.332
P	0.0
PT	0.0
NP	0.0
NPC	0.0
NPTC	0.0

Table 7.3: Percentage of t's variation explained by each effect.

$$\begin{aligned}\ln(t_{ij}) &= 2.22 + 0.602x_T - 0.1748x_N - 0.145x_Nx_T - 0.0903x_Nx_Tx_C \\ &\quad +0.0684x_C + 0.0574x_Tx_C - 0.0566x_Nx_C + 0.0537x_Px_C \\ &\quad +0.0431x_Px_Tx_C - 0.0394x_Nx_Px_T + E_{ij} \end{aligned} \tag{7.3}$$

or equivalently,

$$\begin{aligned}t_{ij} &= 9.21 \times 1.83^{x_T} \times 0.840^{x_N} \times 0.865^{x_Nx_T} \times 0.914^{x_Nx_Tx_C} \times 1.07^{x_C} \\ &\quad \times 1.06^{x_Tx_C} \times 0.945^{x_Nx_C} \times 1.06^{x_Px_C} \times 1.04^{x_Px_Tx_C} \times 0.961^{x_Nx_Px_T} \\ &\quad \times e^{E_{ij}}. \end{aligned} \tag{7.4}$$

Table 7.3 shows the percentage of the variation in the mean response time that is explained by each factor, combination of factors, and experimental error. The entries are ordered by level of importance. The factor T is the most important. It accounts for 77 percent of the total variation. The second most important factor is N. The factor N and the interaction of N and T account for 11 percent of the total variation. All entries labelled as 0.0 percent are not statistically significant at a 90 percent confidence level. These insignificant factors include the primary factor P. Factor C is barely significant; it only explains 1.0 percent of the total variation. The experimental error accounts for less than 6 percent of the total variation. This level is acceptable.

The large effect associated with T, the number of producer tasks on each processor,

points to a bottleneck in the HARTOS implementation. Specifically, there is apparently a large queueing delay associated with the reply and request NP mailbox associated with each AP. This conclusion is arrived at by considering the relative effects. The processors on a given node share the outgoing queue for the network. If there were significant contention at this queue, then the factor P would have had a greater effect on the performance. The fact that P had an insignificant effect is evidence against this hypothesis. In fact, all the queues in the NP are shared by both processors, except for the AP-specific reply and request mailbox queue and the send mailbox queue on each AP. The send mailbox queue may add some delay to the response time. However, the queue length is limited to one because the queued task does a busy wait on the mailbox. Other tasks do not get scheduled and therefore do not have to opportunity to make the remote call until the queued task has been serviced. The time that the task may be queued is bounded from above by the maximum time that the NP requires to execute its polling cycle.

If we look more closely at the effect of N on total variation, we see that the effect, q_N, is negative. This can be seen clearly by comparing the $\bar{t}$ values in Table 7.3. The response times for $N = 2$ are higher than for $N = 4$. This is contrary to what would be expected. It is expected that more nodes on the network would cause greater contention for the Ethernet and thus higher response times. It is possible that this behavior is a result of the combination of the loose method used to synchronize the workloads and the different traffic pattern used in the four node experiments. As discussed above, the looseness of the synchronization may have allowed some processors to begin sending messages at a significantly long time before or after the other processors. These early and late processors would be communicating on an otherwise empty network, and would subsequently have lower response times. Furthermore, the effort to spread out the load on the four node experiments, in order to reduce the variance in response times measured on different processors may have added to this effect. It may be the case that the early and late processors were distributing the message processing load over a number of idle network processors, thus further reducing the response time. The messages from these early or late processors were not even encountering the queuing delays caused by other messages from the same processor.

Verifying the exact cause of this seemingly anomalous behavior can be the subject of further investigation. The necessary experiments will involve measuring response time with different traffic configurations and different synchronization techniques. Studying different traffic configurations will involve simply changing the processor assignments for the different destination tasks and performing the experiments again. Studying the effects of workload synchronization on performance would entail comparing the performance of the

Factor levels T, N, C, P	$\overline{m}$	s_m	$\overline{\ln m}$
1, 2, 1, 1	0.01	0.0224	-0.599
1, 2, 1, 2	0.0	0.0	0.0
1, 2, 10, 1	0.01	0.0224	-0.599
1, 2, 10, 2	0.0	0.0	0.0
1, 4, 1, 1	14.4	4.74	2.61
1, 4, 1, 2	2.23	1.3	0.611
1, 4, 10, 1	5.87	1.31	1.75
1, 4, 10, 2	4.22	0.95	1.42
10, 2, 1, 1	10.4	4.41	2.23
10, 2, 1, 2	14.9	11.5	2.21
10, 2, 10, 1	10.3	3.26	2.28
10, 2, 10, 2	16.2	5.86	2.71
10, 4, 1, 1	50.4	27.2	3.77
10, 4, 1, 2	36.9	6.83	3.6
10, 4, 10, 1	42.7	16.6	3.7
10, 4, 10, 2	50.3	13.5	3.88

Table 7.4: Results for the percentage of lost messages: mean $\overline{m}$, standard deviation s_m, and mean of the transformed data $\overline{\ln m}$.

SW on this version of HARTOS against the performance of the same SW on a version of HARTOS which supported distributed clock synchronization. Such investigations would provide no further illumination of the capabilities of the SWG, and will thus not be pursued here.

Next, we analyze the system based on the percentage of messages lost. The statistics for m are shown in Table 7.4. The values for the mean, $\overline{m}$, and standard deviation, s_m, are shown for reference purposes only. Based on the log-transformed values, we calculate the effects of the various factor combinations. The effects are shown in Table 7.5 along with the 90% confidence intervals. If the confidence interval includes 0, then the effect is insignificant at this confidence level. We note that only six of the factor combinations and the effects of error are significant.

The percentage of the variation of m that is caused by each significant factor combination is shown in Table 7.6. From this data we get the following model for the percentage of messages that are lost.

$$\begin{aligned} \ln(m_{ij}) &= 1.85 + 1.2x_T + 0.818x_N - 0.245x_Nx_P + 0.196x_Nx_Px_T \\ &\quad +0.154x_Px_C - 0.13x_Nx_T + E_{ij} \end{aligned} \tag{7.5}$$

Effect	Value	90% Confidence Interval
q_0	1.85	[1.72, 1.98]
q_T	1.2	[1.07, 1.33]
q_N	0.818	[0.69, 0.946]
q_{NP}	-0.245	[-0.374, -0.117]
q_{NPT}	0.196	[0.0675, 0.324]
q_{PC}	0.154	[0.0259, 0.282]
q_{NT}	-0.13	[-0.258, -0.00185]
q_{NPTC}	-0.11	[-0.238, 0.0183]
q_{NPC}	0.0978	[-0.0304, 0.226]
q_{PT}	0.0976	[-0.0305, 0.226]
q_{PTC}	-0.0535	[-0.182, 0.0746]
q_{TC}	0.0508	[-0.0773, 0.179]
q_C	0.0445	[-0.0836, 0.173]
q_P	-0.0439	[-0.172, 0.0842]
q_{NC}	-0.0236	[-0.152, 0.105]
q_{NTC}	-0.0173	[-0.145, 0.111]

Table 7.5: Effects for the multiplicative model for m.

Factor(s)	Percentage
T	54.0
N	25.1
ERROR	14.1
NP	2.26
NPT	1.44
PC	0.89
NT	0.634
P	0.0
C	0.0
PT	0.0
TC	0.0
NC	0.0
NPC	0.0
PTC	0.0
NTC	0.0
NPTC	0.0

Table 7.6: Percentage of m's variation explained by each effect.

or equivalently,

$$m_{ij} = 6.36 \times 3.32^{x_T} \times 2.27^{x_N} \times 0.783^{x_N x_P} \times 1.21^{x_N x_P x_T} \times 1.17^{x_P x_C} \times 0.878^{x_N x_T} \times e^{E_{ij}}. \quad (7.6)$$

The conclusions drawn for m parallel those for t. The number of tasks sending messages from each processor had the greatest effect. However, for this performance index, the number of nodes, N, had a greater relative effect than it had on t. In fact, the effect is positive, indicating an increased effect due to N regardless of the communication configuration. Experimental error had a greater effect on m than on t. For m, the factors P and C were not significant at a 90 percent confidence level.

Since it is not known why the system loses messages, the three most significant factors should be evaluated further. The high experimental error may indicate the effects of other factors that were not considered in the experiments. The nature of these other factors should also be determined. For users of HARTOS version 1, the high number of lost messages for heavy workloads points to the need for the specification of retries for remote calls. The average values for t may be used as a gauge for the timeout before a retry is attempted. Retries should significantly reduce the number of lost messages.

7.6 Summary and Conclusions

Through these experiments, we demonstrated the ability of the SWG and SW to be used for experimental analysis of distributed, real-time computer systems. We showed how an SW may be specified that demonstrates the particular workload characteristics required for an evaluation. We showed how object templates may be used to produce a reasonably large workload from a compact specification. The workload with the greatest number of tasks and exchanges was generated from a graph file only slightly different from the one shown in Appendix II. It contained 160 tasks and 80 exchanges distributed over eight processors. The size of the workload which can be specified and generated is only restricted by the available system resources. There are no limitations imposed by the SWG. We also showed how, even with a large workload, by using appropriately defined constants, changes to the specification which are necessary to instantiate different workload configurations may be localized.

The results of this evaluation provided some important insights into the performance of HARTOS version 1. The primary conclusion is that the NP-AP interface is a serious bottleneck. Future versions of HARTS and HARTOS must pay special attention

to this issue. Further evaluations of HARTS could be performed to fully characterize the effect of this bottleneck at different levels of T. As with the proposed evaluation of the effects of N, such evaluations would use the same techniques used in this chapter. These techniques have already been demonstrated. The focus of this chapter is demonstrating the capabilities of the SWG, not to provide a full characterization of HARTS. Therefore, further evaluations are not included here.

The evaluation also provided insight into the design and use of the SWG. First, we noted that for multiprocessor workloads, it was more convenient to have the processor assignments for all tasks in a single location in the input files. This observation suggests that a new "processor assignment" section be added to the graph file and all assignments be located there. Such a feature was previously suggested by another researcher in real-time systems [43]. It will be included in future versions of the SWG.

CHAPTER 8

CONCLUSIONS

We have described a complete system for generating SWs for distributed real-time systems. The design was influenced by the desire to accurately represent real-time workloads, to support the process of experimental evaluation, and to be flexible, compact, and easy to use. To achieve these goals, we defined a workload model which describes all aspects of a real-time workload. The model is based on notations used to specify real software. It contains user-specifiable parameters to describe the system specific aspects of the workload. This model is the basis for SWSL, a language for specifying SWs. SWSL includes a number of features to improve its ability to specify SWs and to support experimental evaluation.

The SWSL specification of an SW is compiled by an SWG to produce an executable SW. The SW uses a driver which provides distributed control and support for experimentation. We demonstrated that the SW can accurately represent actual workloads and that SWSL could be used successfully for experimental evaluation.

8.1 Research Contributions

Our work contains a number of contributions to the field. We recognized (what should be an obvious fact) that the workload is composed of the application software which is executing on the system and modeled our SW accordingly. Our model specifies the structure of the workload using a notation based on structured analysis and rapid prototyping notations. We added a number of parameters in our model to describe the system-dependent resource requirements of the workload. We also rejected the standard firing rules defined for dataflow models in favor of a model which specifies the invocation and interactions of tasks based on their scheduling parameters and on the operations they execute.

We were the first to define a high-level language to specify SWs and implement an SWG to compile it. The only other language for which the existence of a compiler is mentioned is APET [4], but APET resembled a simple job control language. It was

extremely limited in its capabilities and was therefore not comparable to SWSL.

An important contribution to the specification of SWs is our specification of object templates. The concept of producing multiple instances of an object from a common template can be traced back to the earliest SWs. It is one of the primary mechanisms for abstracting a workload using a compact notation. Our implementation extends this basic idea to allow full support of the object templates within the task graph. Object templates may be used to specify multiple instances of single objects, or they may be combined to specify multiple instances of entire subgraphs. The syntax for specifying the inputs and outputs of object templatess allow the graph connectivity to be described explicitly. In addition, abbreviated syntax is available for some common combinations of templates. The SWG will automatically expand these abbreviated definitions according to a known set of connection rules.

We have also identified a number of features required to support experimentation using the SW. The most important of these is the multiple run facility. Another comes from the observation that an interactive user interface is not acceptable for SWs for real-time systems. Therefore, all run-time parameters are specified in SWSL and compiled into the SW. Then, the SW may be downloaded to the target embedded system and executed. It will exhibit the specified behaviors without user interference. To the same end, we defined the experiment file which contains the parameters for the experiment. These parameters must also be compiled into the SW.

A number of contributions are exhibited in the functions file. The most useful is the concept of plug-in functions. To support these functions, we defined two synthetic operations, `sread()` and `swrite()`, which use the information provided by the SWG describing the connected objects to select which system call to use to perform the required read or write operation.

We introduced the idea of the verbatim code segments within the functions. By allowing the user to specify the function using both synthetic operations and exact code, we increase the flexibility of SWSL.

We were the first to consider the statistical properties of the SW. We defined a number of parameters that take their values at run-time from random number generators. Control constructs with stochastic behavior are defined in functions to model data-dependent behavior of the task. We base all stochastic activities on separate random number generator streams to improve their statistical independence. We also provide the capability to specify the seeds for the random number streams and to decide whether or not to reset the streams to the specified seed between runs. This feature improves the reproducibility

of experiments.

Finally, we defined the necessary features of an SW driver for a distributed real-time system. It must have distributed control, support for experimentation, and should simulate terminators.

8.2 Future Directions

SWSL was designed to provide a full range of capabilities for specifying synthetic workloads for distributed real-time systems. In the process of writing the SW specifications for the evaluations in Chapters 6 and 7, we discovered a few places where the language could be extended or modified. The robot control software in Chapter 6 was fairly complex. SWSL cannot define procedures to be called from synthetic functions. Therefore, all code for a task must be written in one function specification. SWSL would benefit from the ability to specify procedures using the SWSL function syntax. It would provide the user with the ability to specify commonly executed segments of code in a single procedure instead of having to duplicate that code throughout the function. In addition, recursive procedures should be supported. Within each recursive procedure, the decision of whether or not to recurse would be decided probabilistically, because there are no data values in SWSL upon which the decision can be based. Therefore, the number of levels of recursion will be a random value with a distribution determined by the probability that a given level will recurse. Although procedures that recurse an unpredictable number of times are generally considered inappropriate for hard real-time systems, they may be used in soft real-time systems. Therefore, SWSL should be able to specify recursive procedures which may be used to evaluate soft real-time systems.

Two modifications to SWSL were suggested by the requirements of the evaluation in Chapter 7. First is the ability to specify the processor assignments of the SW objects in a single location in the input files. This feature was discussed previously. The second feature is the ability to specify object templates such that multiple instances of the object exist on the same processor. Such a feature would have reduced the size of the graph specification in Appendix H to only two transformation definitions (one for the producer and one for the consumer) and one store definition. If multiple instances of the same object could exist on the same processor, then the SWSL specification of `INPUTs` and `OUTPUTs` would have to include a specification of which copy of the object on each processor. Adding this information to the specification would make the specification of `INPUTs` and `OUTPUTs` more complex. An alternative is to change the specification of `INPUTs` and `OUTPUTs` to use an

indexing notation. The various instances of an object would be located on the processors in the PROCESSOR parameter specification. A given processor label could be listed multiple times in the PROCESSOR parameter list, indicating multiple instances of the object on that processor. The position of the processor labels in the list would provide an index by which the various objects could be referenced. The INPUT and OUTPUT specifications could use this index to specify connected objects. Shortened notations would allow the user to specify flows between instances of objects by indicating the relative positions of the instances in the list.

These modifications to the language would improve the convenience with which the language may be used, but they would not add any modeling power to the language. SWSL is sufficiently powerful as defined. We found no situation where it was not able to specify the required properties.

The modifications listed above are minor changes to the existing SWSL. There is at least one area where more significant research could be applied. SWSL could be modified to define transformations with multiple threads of control. Our workload model defines tasks with single threads of control. A number of current experimental real-time operating systems support multiple threads of control within tasks. Workloads that execute on such systems must be characterized and the appropriate changes made to the workload model and SWSL. A more extensive change would be to explore the use of the object-oriented software paradigm. The particular characteristics of object-oriented workloads should be studied. The findings could be incorporated into the workload model, or, if necessary, a new workload model could be developed.

Another area for future research is the study of the use of SWs. We framed our discussion of SWs in the context of two types of evaluations: those that use SWs that are representative of actual applications, and those that use SWs that produce specific, user-controlled resource demands, but are not necessarily representative of any specific workload. Currently, many researchers are interested in SWs for the first type of evaluation. They want to show that their experimental system meets the requirements of a real-time application. While representative SWs are good for demonstrating this fact, these evaluations do not necessarily provide a quantitative measure of the performance of the system. A collection of representative SWs allow the user to measure the system while it executes a few sample workloads, but they do not provide a rigorous characterization of the system.

The second type of evaluation is intended to produce this rigorous characterization. They allow the user to measure the system under various combinations of workload characteristics and make statistically valid evaluations of the system based on these measurements. We consider these evaluations to be the more important application of the SW.

To support these evaluations, further research should be pursued to improve the ability of SWSL to specify the resource demands of the SW. The resource demands are produced primarily by the synthetic operations in the library. Continued research should be focused on creating a library of operations that contains a wide variety of monoresource synthetic operations with orthogonal resource requirements. Using monoresource synthetic operations simplifies the process of specifying an SW which contains those operations. If the synthetic operations perform complex functions, then the user must spend a great deal of time selecting the proper operation and then choosing the proper parameters with which to invoke the operation. If each synthetic operation exercises a single resource, then the user can select exactly the operation required. If the resource requirements of the operations are orthogonal, then only a single operation will exercise a given resource in a particular manner. The user will not have to decide between multiple choices when selecting which operations to use, and constructing SWs will become much simpler. If more complex behaviors are required for operations, then the user should model the complex operation using a separate transformation with the behavior specified as the transformation's ϕ function. The function can then be archived and used as a plug-in function for other SWs where the same behavior is required.

The creation of an archive of plug-in functions may be the solution to the problem of how to build representative SWs. In Chapter 6, we demonstrated that the SWG is capable of generating representative SWs. We were able to make this determination because we had an existing real-time workload that we could use as the basis of the SW specification. Then we could compare the performance of the SW with that of the workload to determine the representativeness of the SW. This approach to creating a representative SW is not usually possible in practice. Often, researchers desire a representative SW because they have no real application. If they had a real application, they would use it for the evaluation. Without a real application, the representativeness of an SW can not be verified. However, we may be able to construct representative SWs without having a complete, real application with which to compare.

We hypothesize that a representative SW could be constructed from plug-in functions which have been verified to be representative of individual functions in real workloads. This hypothesis is based on observations of the trend toward reusable software in the field of software engineering. Software engineers are attempting to build new software systems based on components developed for previous software systems. If real-time software follows this trend, then real-time workloads will be composed of several reused components to perform standard functions and custom-written components to perform all other functions.

SWs to model these workloads could be constructed from archived plug-in functions which have been verified to represent the reused components. The user would only have to write the SWSL functions to model the behavior of the custom components in the workload. The selection of archived and custom components could be based on a high-level specification of the workload such as the requirements model or the structured analysis specification. SWs constructed in this manner should closely approximate the behavior of the real workload. Further research must be conducted to confirm or refute this hypothesis.

APPENDIX A

SWSL GRAMMAR

The following is the BNF for the grammar which describes SWSL. We give the grammars for the graph file, the experiment file, and the function file. Terminals either represent keywords or specific symbols and are shown as those keywords in uppercase or as the symbols. The regular expressions used to define some terminals are in the exact format as in the lexical analyser generator input file. Nonterminals are shown in lowercase. The empty terminal is denoted with an epsilon, ϵ.

A.0.1 Graph File Grammar

program	::=	GRAPH graph
graph	::=	declarations DEFINITIONS obj_defs
declarations	::=	externs constants OBJECTS obj_decls
externs	::=	EXTERNS externdefs
	|	EXTERNS
	|	ϵ
externdefs	::=	extern_def ;
	|	externdefs extern_def ;
extern_def	::=	extern_type extern_name
	|	extern_def , extern_name
extern_type	::=	identifier
extern_name	::=	identifier
constants	::=	CONSTANTS cons
	|	CONSTANTS
	|	ϵ
cons	::=	const ;

	|	cons const ;
const	::=	cons_name = expr
cons_name	::=	identifier
object_decl_list	::=	obj_decl ;
	|	object_decl_list obj_decl ;
obj_decl	::=	object_type object_name
	|	obj_decl , object_name
object_name	::=	identifier
object_type	::=	identifier
object_def_list	::=	obj_def ;
	|	object_def_list obj_def ;
obj_def	::=	object_name [parameter_list]
parameter_list	::=	parm ;
	|	parameter_list parm ;
parm	::=	parameter
	|	io_parameter
parameter	::=	parameter_name = expr
	|	parameter , expr
parameter_name	::=	identifier
io_parameter	::=	io_parameter_name = io_list
io_parameter_name	::=	INPUT
	|	OUTPUT
io_list	::=	src_dst_name location flow_type
	|	io_list | io_name flow_type
flow_type	::=	: identifier
	|	ϵ
src_dst_name	::=	identifier
location	::=	. processor_name io_number
	|	ϵ
processor_name	::=	identifier
	|	ϵ
io_number	::=	. number

	|	ϵ
expr	::=	expr + expr
	|	expr − expr
	|	expr * expr
	|	expr / expr
	|	− expr
	|	(expr)
	|	value
	|	distribution_name (parmlist)
value	::=	identifier
	|	number
	|	string
parmlist	::=	expr
	|	parmlist , expr
	|	ϵ
distribution_name	::=	identifier
identifier	::=	`[a-zA-Z_][a-zA-Z0-9_]*`
string	::=	`"[^"]*"`
number	::=	integer
	|	float
integer	::=	`"0x"?[0-9]+`
float	::=	`[0-9]+"."[0-9]*`

A.0.2 Experiment File Grammar

program	::=	EXPERIMENT experiment
experiment	::=	constants PARAMETERS exp_definition_list
exp_definition_list	::=	exp_definition ;
	|	exp_definition_list exp_definition ;
exp_definition	::=	exp_name [exp_parameter_list]

```
exp_name              ::=  identifier

exp_parameter_list    ::=  exp_parameter ;
                      |    exp_parameter_list exp_parameter ;
exp_parameter         ::=  exp_parameter_name = expr
                      |    exp_parameter , expr
exp_parameter_name    ::=  identifier
```

A.0.3 Function File Grammar

```
program               ::=  FUNCTIONS functions
functions             ::=  externs constants function_code
function_code         ::=  CODE function_list
function_list         ::=  function_list function
                      |    function
function              ::=  identifier { io_declarations code } ;
io_declarations       ::=  io_declaration ;
                      |    io_declarations io_declaration ;
io_declaration        ::=  io_parameter_name = func_io_list
                      |    io_declaration , func_io_list
func_io_list          ::=  func_io_name func_flow_type
func_flow_type        ::=  : identifier
func_io_name          ::=  identifier

code                  ::=  BEGIN stmtlist END ;
stmtlist              ::=  statementlist
                      |    ϵ
statementlist         ::=  statementlist statement ;
                      |    statement ;
statement             ::=  loop_statement
                      |    switch_statement
                      |    simple_statement

loop_statement        ::=  LOOP loopbound { loop_body }
```

```
loopbound          ::=  expr
                   |    FOREVER
loop_body          ::=  statementlist

switch_statement   ::=  SWITCH { case_list }
case_list          ::=  case_list case ;
                   |    case ;
case               ::=  value : { statementlist }

simple_statement   ::=  identifier ( parmlist )
```

APPENDIX B

ROBOT CONTROL SW SPECIFICATION: GRAPH FILE

```
GRAPH

EXTERNS

FUNC server_func;
FUNC periodic_func;
PROC expos;

CONSTANTS

AuxClock_ELEMENT_SIZE = 4;
adc_board_in_ELEMENT_SIZE = 4;
adc_board_in_rate = 2;
adc_board_out_ELEMENT_SIZE = 4;
adc_board_out_rate = 2;
ai_board_in_rate = 2;
ai_board_in_size = 4;
ai_board_out_rate = 2;
ai_board_out_size = 4;
buf_rec_size = 20;
client_mess_size = 10;
client_rate = 600;
command_ELEMENT_SIZE = 4;
console_ELEMENT_SIZE = 4;
console_rate = 2;
control_ELEMENT_SIZE = 4;
dac_board_in_ELEMENT_SIZE = 4;
dac_board_in_rate = 2;
dac_board_out_ELEMENT_SIZE = 4;
dac_board_out_rate = 2;
file_system_ELEMENT_SIZE = 4;
file_system_rate = 2;
globals_ELEMENT_SIZE = 4;
motor_ELEMENT_SIZE = 4;
rt_period = 1;
rt_priority = 2;
sensors_ELEMENT_SIZE = 4;
server_priority = 80;
status_ELEMENT_SIZE = 4;

include vxworks.constants

OBJECTS

TRANS server;
TRANS rt;
STORE ring_buffer;
TERM client;
TERM ai_board_out;
TERM ai_board_in;
TERM adc_board_out;
TERM adc_board_in;
TERM dac_board_out;
TERM file_system;
TERM console;
STORE command_semaphore;
STORE dac_semaphore;
STORE adc_semaphore;
STORE sensors_semaphore;
STORE command;
STORE sensors;
STORE AuxClock;
STORE control;
STORE motor;
STORE status;
STORE globals;

DEFINITIONS

/*TRANS*/ server [
  SPORADIC = 0;
  FUNCTION = server_func();
include server_io
  START_TIME = 1;
  PROCESSOR = expos;
  ACTIVE = true;
  PRIORITY = server_priority;
  NAME = "serv";
];

/*TRANS*/ rt [
  PERIOD = rt_period;
```

```
  FUNCTION = periodic_func();
  START_TIME = 1;
include rt_io
  PROCESSOR = expos;
  ACTIVE = true;
  DEADLINE = rt_period;
  PRIORITY = rt_priority;
  NAME = "lvl0";
];

/*STORE*/ ring_buffer [
  name = "queue";
  TYPE = depletable;
  ELEMENT_SIZE = buf_rec_size;
  INPUT = server : discrete;
  OUTPUT = rt : discrete;
  PROCESSOR = expos;
  CAPACITY = NOLIMIT;
  ACCESS = all;
  POLICY = fifo;
];

/*TERM*/ client[
  NAME = "client";
  OUTPUT = server : discrete;
  TYPE = source;
  ELEMENT_SIZE = client_mess_size;
  PROCESSOR = expos;
  RATE = client_rate;
  START_TIME = 1;
];

/*TERM*/ ai_board_out[
  NAME = "ai_out";
  INPUT = rt : discrete;
  INPUT = server : discrete;
  TYPE = Sink;
  ELEMENT_SIZE = ai_board_out_size;
  PROCESSOR = expos;
  RATE = ai_board_out_rate;
  START_TIME = 1;
];

/*TERM*/ ai_board_in[
  NAME = "ai_in";
  OUTPUT = rt : continuous;
  OUTPUT = server : continuous;
  TYPE = Source;
  ELEMENT_SIZE = ai_board_in_size;
  PROCESSOR = expos;
  RATE = ai_board_in_rate;
  START_TIME = 1;
];

/*TERM*/ adc_board_out[
  NAME = "adc_out";
  INPUT = rt : discrete;
  TYPE = Sink;
  ELEMENT_SIZE =
adc_board_out_ELEMENT_SIZE;
  PROCESSOR = expos;
  RATE = adc_board_out_rate;
  START_TIME = 1;
];

/*TERM*/ adc_board_in[
  NAME = "adc_in";
  OUTPUT = rt : continuous;
  OUTPUT = server : continuous;
  TYPE = Source;
  ELEMENT_SIZE =
adc_board_in_ELEMENT_SIZE;
  PROCESSOR = expos;
  RATE = adc_board_in_rate;
  START_TIME = 1;
];

/*TERM*/ dac_board_out[
  NAME = "dac_out";
  INPUT = rt : discrete;
  INPUT = server : discrete;
  TYPE = Sink;
  ELEMENT_SIZE =
dac_board_out_ELEMENT_SIZE;
  PROCESSOR = expos;
  RATE = dac_board_out_rate;
  START_TIME = 1;
];

/*STORE*/ command_semaphore[
  name = "com_sem";
  TYPE = DEPLETABLE;
  ELEMENT_SIZE = 0;
  INPUT = rt : discrete;
  OUTPUT = rt : discrete;
  PROCESSOR = expos;
  CAPACITY = 0;
  ACCESS = exclusive;
];

/*STORE*/ dac_semaphore[
  name = "dac_sem";
  TYPE = DEPLETABLE;
  ELEMENT_SIZE = 0;
  INPUT = rt : discrete;
  INPUT = server : discrete;
```

```
  OUTPUT = rt : discrete;
  OUTPUT = server : discrete;
  PROCESSOR = expos;
  CAPACITY = 0;
  ACCESS = exclusive;
];

/*STORE*/ adc_semaphore[
  name = "adc_sem";
  TYPE = DEPLETABLE;
  ELEMENT_SIZE = 0;
  INPUT = rt : discrete;
  INPUT = server : discrete;
  OUTPUT = rt : discrete;
  OUTPUT = server : discrete;
  PROCESSOR = expos;
  CAPACITY = 0;
  ACCESS = exclusive;
];

/*STORE*/ sensors_semaphore[
  name = "sens_sem";
  TYPE = DEPLETABLE;
  ELEMENT_SIZE = 0;
  INPUT = rt : discrete;
  OUTPUT = rt : discrete;
  PROCESSOR = expos;
  CAPACITY = 0;
  ACCESS = exclusive;
];

/*STORE*/ command[
  name = "comm";
  TYPE = NONDEPLETABLE;
  ELEMENT_SIZE =
command_ELEMENT_SIZE;
  INPUT = rt : discrete;
  INPUT = server : discrete;
  PROCESSOR = expos;
  CAPACITY = 1;
  ACCESS = all;
];

/*STORE*/ sensors[
  name = "sens";
  TYPE = NONDEPLETABLE;
  ELEMENT_SIZE =
sensors_ELEMENT_SIZE;
  INPUT = rt : discrete;
  INPUT = server : discrete;
  OUTPUT = server : continuous;
  PROCESSOR = expos;
  CAPACITY = 1;
  ACCESS = all;
];

/*STORE*/ AuxClock[
  name = "auxclk";
  TYPE = NONDEPLETABLE;
  ELEMENT_SIZE =
AuxClock_ELEMENT_SIZE;
  INPUT = server : discrete;
  PROCESSOR = expos;
  CAPACITY = 1;
  ACCESS = all;
];

/*STORE*/ control[
  name = "control";
  TYPE = NONDEPLETABLE;
  ELEMENT_SIZE =
control_ELEMENT_SIZE;
  INPUT = server : discrete;
  INPUT = rt : discrete;
  OUTPUT = server : continuous;
  PROCESSOR = expos;
  CAPACITY = 1;
  ACCESS = all;
];

/*STORE*/ motor[
  name = "motor";
  TYPE = NONDEPLETABLE;
  ELEMENT_SIZE = motor_ELEMENT_SIZE;
  OUTPUT = server : continuous;
  PROCESSOR = expos;
  CAPACITY = 1;
  ACCESS = all;
];

/*TERM*/ file_system[
  NAME = "file";
  INPUT = server : discrete;
  TYPE = Sink;
  ELEMENT_SIZE =
file_system_ELEMENT_SIZE;
  PROCESSOR = expos;
  RATE = file_system_rate;
  START_TIME = 1;
];

/*TERM*/ console[
  NAME = "console";
  INPUT = server : discrete;
  TYPE = Sink;
```

```
  ELEMENT_SIZE =
console_ELEMENT_SIZE;
  PROCESSOR = expos;
  RATE = console_rate;
  START_TIME = 1;
];

/*STORE*/ status[
  name = "status";
  TYPE = NONDEPLETABLE;
  ELEMENT_SIZE = status_ELEMENT_SIZE;
  INPUT = rt : discrete;
  INPUT = server : discrete;
  OUTPUT = rt : continuous;
  PROCESSOR = expos;
  CAPACITY = 1;
  ACCESS = all;
];

/*STORE*/ globals[
  name = "globals";
  TYPE = NONDEPLETABLE;
  ELEMENT_SIZE =
globals_ELEMENT_SIZE;
  INPUT = rt : discrete;
  INPUT = server : discrete;
  OUTPUT = rt : continuous;
  OUTPUT = server : continuous;
  PROCESSOR = expos;
  CAPACITY = 1;
  ACCESS = all;
];
```

APPENDIX C

ROBOT CONTROL SYNTHETIC WORKLOAD SPECIFICATION: EXPERIMENT FILE

```
EXPERIMENT

CONSTANTS

Runs = 4;

PARAMETERS

default[
  TIMEUNIT = 1000/60; /* 60 Hz */
  TIMING = true;
  TIMELIMIT = 5000;
  SEED = 727633698, 276090261,1808217256;
  SEED_RESET = TRUE, TRUE, TRUE;
];
```

APPENDIX D

ROBOT CONTROL SYNTHETIC WORKLOAD SPECIFICATION: FUNCTION FILE

FUNCTIONS

EXTERNS

DIST expon;

OPER swrite;
DIST intvalue; /* *intvalue(x) returns (int)x* */
OPER sread;
OPER bool;
OPER trig;
OPER fp;

CONSTANTS

rare = 1;
fp_scaling_factor = 0.70; /* *Value for SW1: 1.0* */

CODE

periodic_func {
verbatim
`#include "probe.h"`
endverbatim
include rt_io

```
BEGIN

verbatim
probe(1);
endverbatim
  swrite(adc_board_out);
  swrite(ai_board_out);

  sread(command_semaphore, WAIT);
  LOOP (18+6) { /* loop counts and expressions in fp() calls were derived
                   from the robot control software */
    swrite(command);
  };

  swrite(command_semaphore);
  SWITCH {
    rare : { /* if stopped */
        sread(ring_buffer, NOWAIT);
        SWITCH {
          1/600 : { /* if a message was read */
            SWITCH {
              33 : { /* grasp command? */
                swrite(status);
              };
              remaining : {
                SWITCH {
                  0 : { /* relative move command */
                    fp(intvalue(fp_scaling_factor * (6)));
                  };
                };
              };
            };
          };
        };
```

```
    };
    fp(intvalue(fp_scaling_factor * (6*6+5+6*3)));
  };
  remaining : {
    SWITCH {
      96 : { /* constant velocity */
        swrite(control);
        swrite(globals);
      };
    };
    SWITCH {
      4 /* 75 */ : { /* same segment? Value for SW1: 75 */
        sread(ring_buffer, NOWAIT);
        SWITCH {
          1/600 : { /* if a message was read */
            SWITCH {
              33 : { /* grasp command? */
                swrite(status);
              };
              remaining : {
                SWITCH {
                  0 : { /* relative move command? */
                    fp(intvalue(fp_scaling_factor * (6)));
                  };
                };
              };
            };
          };
          SWITCH {
            99 : { /* no new data? */
              swrite(globals);
            };
          };
          fp(intvalue(fp_scaling_factor * (6*6+5+6*3)));
        };
```

```
        remaining : { /* in new segment */
          SWITCH {
            33 : { /* decelerating? */
              swrite(control);
            };
            remaining : {
              swrite(globals);
            };
          };
        };
      };
    };
  };
};
fp(intvalue(fp_scaling_factor * (6)));
SWITCH { /* sensor type */
  100 : { /* encoder */
    LOOP 6 {
      sread(ai_board_in, NOWAIT);
      bool(1);
    };
    fp(intvalue(fp_scaling_factor * (6)));
  };
  remaining : { /* pots */
    LOOP 60 {
      sread(adc_semaphore, WAIT);
      sread(adc_board_in, NOWAIT);
      swrite(adc_semaphore);
    };
    fp(intvalue(fp_scaling_factor * (6)));
  };
};
fp(intvalue(fp_scaling_factor * (12)));
trig(12);
fp(intvalue(fp_scaling_factor * (6)));
```

```
SWITCH {
  100 : { /* force used */
    LOOP 6 {
      sread(adc_semaphore, WAIT);
      sread(adc_board_in, NOWAIT);
      swrite(adc_semaphore);
    };
    fp(intvalue(fp_scaling_factor * (2*36+6)));
    trig(2);
    fp(intvalue(fp_scaling_factor * (12+4)));
    trig(2);
    fp(intvalue(fp_scaling_factor * (12+4)));
  };
};
sread(sensors_semaphore, WAIT);
LOOP 6 {
  swrite(sensors);
};
swrite(sensors_semaphore);
SWITCH { /* check mode */
  100 : { /* position interpolated */
    fp(intvalue(fp_scaling_factor * (12*4+6*5)));
    swrite(ai_board_out);
  };
  rare : { /* torque */
    fp(intvalue(fp_scaling_factor * (6*3)));
    LOOP 6 {
      sread(dac_semaphore, WAIT);
      swrite(dac_board_out);
      swrite(dac_semaphore);
    };
    swrite(ai_board_out);
  };
  rare : { /* standby */
    fp(intvalue(fp_scaling_factor * (12)));
```

```
    };
  };
  sread(adc_semaphore, WAIT);
  swrite(adc_board_out);
  swrite(adc_semaphore);
verbatim
probe(2);
endverbatim
END;
};

server_func {
include server_io
BEGIN

LOOP forever {
  sread(client,WAIT);
  SWITCH {
    0 : { /* relative move */
      SWITCH {
        rare : {}; /* if in standby mode */
        remaining : {
          sread(globals, NOWAIT);
          sread(control, NOWAIT);
          fp(intvalue(fp_scaling_factor * (6)));
          sread(control, NOWAIT);
          fp(intvalue(fp_scaling_factor * (6)));
          swrite(ring_buffer);
        };
      };
    };
    33 : { /* absolute move */
      SWITCH {
        rare : {}; /* if in standby mode */
        remaining : {
```

```
            swrite(globals);
            sread(control, NOWAIT);
            fp(intvalue(fp_scaling_factor * (6)));
            sread(control, NOWAIT);
            fp(intvalue(fp_scaling_factor * (6)));
            swrite(ring_buffer);
          };
        };
      };
      33: { /* gripper OPEN */
        swrite(ring_buffer);
      };
      33: { /* gripper CLOSE */
        swrite(ring_buffer);
      };
    };
  };
END;
};
```

APPENDIX E

ROBOT CONTROL SYNTHETIC WORKLOAD SPECIFICATION: INCLUDED FILE "VXWORKS.CONSTANTS"

```
OPERATING_SYSTEM = "VXWORKS" ;

WAIT = 0;
NOWAIT = 1;
NOLIMIT = 50;
RUNNING = 0;
ALL = 0;
GROUPONLY = 1;
FIFO = 0;
PRIO = 0x80;
INFINITE = 0;
LIMITED = 2;
```

APPENDIX F

ROBOT CONTROL SYNTHETIC WORKLOAD SPECIFICATION: INCLUDED FILE "RT_IO"

This include file contains the `INPUT` and `OUTPUT` specifications for the transformation `rt` which implements the level0 task. This file is included into both the transformation specification and the function specification. The syntax is the same for both, and using an include file keeps the specifications consistent.

INPUT = ring_buffer : discrete;
INPUT = ai_board_in : continuous;
INPUT = dac_semaphore : discrete;
INPUT = adc_board_in : continuous;
INPUT = adc_semaphore : discrete;
INPUT = command_semaphore : discrete;
INPUT = sensors_semaphore : discrete;
INPUT = status : continuous;
INPUT = globals : continuous;
OUTPUT = dac_board_out : discrete;
OUTPUT = dac_semaphore : discrete;
OUTPUT = ai_board_out : discrete;
OUTPUT = adc_board_out : discrete;
OUTPUT = adc_semaphore : discrete;
OUTPUT = command_semaphore : discrete;
OUTPUT = command : discrete;
OUTPUT = sensors_semaphore : discrete;
OUTPUT = sensors : discrete;
OUTPUT = status : discrete;
OUTPUT = globals : discrete;
OUTPUT = control : discrete;

APPENDIX G

ROBOT CONTROL SYNTHETIC WORKLOAD SPECIFICATION: INCLUDED FILE "SERVER_IO"

This include file contains the INPUT and OUTPUT specifications for the transformation server. This file is included into both the transformation specification and the function specification. The syntax is the same for both, and using an include file keeps the specifications consistent.

```
INPUT = client : discrete;
INPUT = ai_board_in : continuous;
INPUT = adc_board_in : continuous;
INPUT = dac_semaphore : discrete;
INPUT = adc_semaphore : discrete;
INPUT = sensors : continuous;
INPUT = control : continuous;
INPUT = motor : continuous;
INPUT = globals : continuous;
OUTPUT = console : discrete;
OUTPUT = dac_board_out : discrete;
OUTPUT = dac_semaphore : discrete;
OUTPUT = ai_board_out : discrete;
OUTPUT = adc_semaphore : discrete;
OUTPUT = ring_buffer : discrete;
OUTPUT = command : discrete;
OUTPUT = sensors : discrete;
OUTPUT = AuxClock : discrete;
OUTPUT = control : discrete;
OUTPUT = file_system : discrete;
OUTPUT = status : discrete;
OUTPUT = globals : discrete;
```

APPENDIX H

WORKLOAD (4, 1, 10, 10): GRAPH FILE

```
GRAPH

EXTERNS

FUNC fun2;
FUNC fun1;
PROC rtcl61;
PROC rtcl81;
PROC rtcl91, rtclA1;

CONSTANTS

num_tasks = 10;
loopcount = 10;
per1 = loopcount * num_tasks + 1;
dead1 = per1 - 1;
source_priority = 20;
dest_priority = 10;

source1 = rtcl61;
dest1_1 = rtcl81;
dest1_2 = rtcl91;
dest1_3 = rtclA1;
dest1_4 = rtcl81;
dest1_5 = rtcl91;
dest1_6 = rtclA1;
dest1_7 = rtcl81;
dest1_8 = rtcl91;
dest1_9 = rtclA1;
dest1_10 = rtcl81;

source2 = rtcl81;
dest2_1 = rtcl91;
dest2_2 = rtclA1;
dest2_3 = rtcl61;
dest2_4 = rtcl91;
dest2_5 = rtclA1;
dest2_6 = rtcl61;
dest2_7 = rtcl91;
dest2_8 = rtclA1;
dest2_9 = rtcl61;
dest2_10 = rtcl91;

source3 = rtcl91;
dest3_1 = rtclA1;
dest3_2 = rtcl61;
dest3_3 = rtcl81;
dest3_4 = rtclA1;
dest3_5 = rtcl61;
dest3_6 = rtcl81;
dest3_7 = rtclA1;
dest3_8 = rtcl61;
dest3_9 = rtcl81;
dest3_10 = rtclA1;

source4 = rtclA1;
dest4_1 = rtcl61;
dest4_2 = rtcl81;
dest4_3 = rtcl91;
dest4_4 = rtcl61;
dest4_5 = rtcl81;
dest4_6 = rtcl91;
dest4_7 = rtcl61;
dest4_8 = rtcl81;
dest4_9 = rtcl91;
dest4_10 = rtcl61;

include psos.constants

OBJECTS

TRANS task1;
STORE store1;
TRANS task2;
TRANS task1a;
STORE store1a;
TRANS task2a;
TRANS task1b;
STORE store1b;
TRANS task2b;
TRANS task1c;
STORE store1c;
```

```
TRANS task2c;
TRANS task1d;
STORE store1d;
TRANS task2d;
TRANS task1e;
STORE store1e;
TRANS task2e;
TRANS task1f;
STORE store1f;
TRANS task2f;
TRANS task1g;
STORE store1g;
TRANS task2g;
TRANS task1h;
STORE store1h;
TRANS task2h;
TRANS task1i;
STORE store1i;
TRANS task2i;

DEFINITIONS

/*TRANS*/ task1[
  PERIOD = per1;
  FUNCTION = fun1();
  START_TIME = 1;
  PRIORITY = source_priority;
  OUTPUT = store1 : discrete;
  PROCESSOR = source1, source2, source3,
source4;
  ACTIVE = true;
  DEADLINE = dead1;
];

/*STORE*/ store1[
  TYPE = depletable;
  ELEMENT_SIZE = 10;
  INPUT = task1 : discrete;
  OUTPUT = task2 : discrete;
  PROCESSOR = dest1_1, dest2_1, dest3_1,
dest4_1;
  policy = prio;
];

/*TRANS*/ task2[
  SPORADIC = 0;
  FUNCTION = fun2();
  START_TIME = 1;
  PRIORITY = dest_priority;
  INPUT = store1 : discrete;
  PROCESSOR = dest1_1, dest2_1, dest3_1,
dest4_1;
  ACTIVE = true;
];

/*TRANS*/ task1a[
  PERIOD = per1;
  FUNCTION = fun1();
  START_TIME = 1;
  PRIORITY = source_priority;
  OUTPUT = store1a : discrete;
  PROCESSOR = source1, source2, source3,
source4;
  ACTIVE = true;
  DEADLINE = dead1;
];

/*STORE*/ store1a[
  TYPE = depletable;
  ELEMENT_SIZE = 10;
  INPUT = task1a : discrete;
  OUTPUT = task2a : discrete;
  PROCESSOR = dest1_2, dest2_2, dest3_2,
dest4_2;
  policy = prio;
];

/*TRANS*/ task2a[
  SPORADIC = 0;
  FUNCTION = fun2();
  START_TIME = 1;
  PRIORITY = dest_priority;
  INPUT = store1a : discrete;
  PROCESSOR = dest1_2, dest2_2, dest3_2,
dest4_2;
  ACTIVE = true;
];

/*TRANS*/ task1b[
  PERIOD = per1;
  FUNCTION = fun1();
  START_TIME = 1;
  PRIORITY = source_priority;
  OUTPUT = store1b : discrete;
  PROCESSOR = source1, source2, source3,
source4;
  ACTIVE = true;
  DEADLINE = dead1;
];

/*STORE*/ store1b[
  TYPE = depletable;
  ELEMENT_SIZE = 10;
  INPUT = task1b : discrete;
  OUTPUT = task2b : discrete;
```

```
  PROCESSOR = dest1_3, dest2_3, dest3_3,
dest4_3;
  policy = prio;
];

/*TRANS*/ task2b[
  SPORADIC = 0;
  FUNCTION = fun2();
  START_TIME = 1;
  PRIORITY = dest_priority;
  INPUT = store1b : discrete;
  PROCESSOR = dest1_3, dest2_3, dest3_3,
dest4_3;
  ACTIVE = true;
];

/*TRANS*/ task1c[
  PERIOD = per1;
  FUNCTION = fun1();
  START_TIME = 1;
  PRIORITY = source_priority;
  OUTPUT = store1c : discrete;
  PROCESSOR = source1, source2, source3,
source4;
  ACTIVE = true;
  DEADLINE = dead1;
];

/*STORE*/ store1c[
  TYPE = depletable;
  ELEMENT_SIZE = 10;
  INPUT = task1c : discrete;
  OUTPUT = task2c : discrete;
  PROCESSOR = dest1_4, dest2_4, dest3_4,
dest4_4;
  policy = prio;
];

/*TRANS*/ task2c[
  SPORADIC = 0;
  FUNCTION = fun2();
  START_TIME = 1;
  PRIORITY = dest_priority;
  INPUT = store1c : discrete;
  PROCESSOR = dest1_4, dest2_4, dest3_4,
dest4_4;
  ACTIVE = true;
];
/*TRANS*/ task1d[
  PERIOD = per1;
  FUNCTION = fun1();
  START_TIME = 1;
  PRIORITY = source_priority;
  OUTPUT = store1d : discrete;
  PROCESSOR = source1, source2, source3,
source4;
  ACTIVE = true;
  DEADLINE = dead1;
];

/*STORE*/ store1d[
  TYPE = depletable;
  ELEMENT_SIZE = 10;
  INPUT = task1d : discrete;
  OUTPUT = task2d : discrete;
  PROCESSOR = dest1_5, dest2_5, dest3_5,
dest4_5;
  policy = prio;
];

/*TRANS*/ task2d[
  SPORADIC = 0;
  FUNCTION = fun2();
  START_TIME = 1;
  PRIORITY = dest_priority;
  INPUT = store1d : discrete;
  PROCESSOR = dest1_5, dest2_5, dest3_5,
dest4_5;
  ACTIVE = true;
];
/*TRANS*/ task1e[
  PERIOD = per1;
  FUNCTION = fun1();
  START_TIME = 1;
  PRIORITY = source_priority;
  OUTPUT = store1e : discrete;
  PROCESSOR = source1, source2, source3,
source4;
  ACTIVE = true;
  DEADLINE = dead1;
};

/*STORE*/ store1e[
  TYPE = depletable;
  ELEMENT_SIZE = 10;
  INPUT = task1e : discrete;
  OUTPUT = task2e : discrete;
  PROCESSOR = dest1_6, dest2_6, dest3_6,
dest4_6;
  policy = prio;
];

/*TRANS*/ task2e[
  SPORADIC = 0;
  FUNCTION = fun2();
  START_TIME = 1;
```

```
  PRIORITY = dest_priority;
  INPUT = storele : discrete;
  PROCESSOR = dest1_6, dest2_6, dest3_6,
dest4_6;
  ACTIVE = true;
];
/*TRANS*/ task1f[
  PERIOD = per1;
  FUNCTION = fun1();
  START_TIME = 1;
  PRIORITY = source_priority;
  OUTPUT = store1f : discrete;
  PROCESSOR = source1, source2, source3,
source4;
  ACTIVE = true;
  DEADLINE = dead1;
];

/*STORE*/ store1f[
  TYPE = depletable;
  ELEMENT_SIZE = 10;
  INPUT = task1f : discrete;
  OUTPUT = task2f : discrete;
  PROCESSOR = dest1_7, dest2_7, dest3_7,
dest4_7;
  policy = prio;
];

/*TRANS*/ task2f[
  SPORADIC = 0;
  FUNCTION = fun2();
  START_TIME = 1;
  PRIORITY = dest_priority;
  INPUT = store1f : discrete;
  PROCESSOR = dest1_7, dest2_7, dest3_7,
dest4_7;
  ACTIVE = true;
];

/*TRANS*/ task1g[
  PERIOD = per1;
  FUNCTION = fun1();
  START_TIME = 1;
  PRIORITY = source_priority;
  OUTPUT = store1g : discrete;
  PROCESSOR = source1, source2, source3,
source4;
  ACTIVE = true;
  DEADLINE = dead1;
];

/*STORE*/ store1g[
  TYPE = depletable;
  ELEMENT_SIZE = 10;
  INPUT = task1g : discrete;
  OUTPUT = task2g : discrete;
  PROCESSOR = dest1_8, dest2_8, dest3_8,
dest4_8;
  policy = prio;
];

/*TRANS*/ task2g[
  SPORADIC = 0;
  FUNCTION = fun2();
  START_TIME = 1;
  PRIORITY = dest_priority;
  INPUT = store1g : discrete;
  PROCESSOR = dest1_8, dest2_8, dest3_8,
dest4_8;
  ACTIVE = true;
];
/*TRANS*/ task1h[
  PERIOD = per1;
  FUNCTION = fun1();
  START_TIME = 1;
  PRIORITY = source_priority;
  OUTPUT = store1h : discrete;
  PROCESSOR = source1, source2, source3,
source4;
  ACTIVE = true;
  DEADLINE = dead1;
];

/*STORE*/ store1h[
  TYPE = depletable;
  ELEMENT_SIZE = 10;
  INPUT = task1h : discrete;
  OUTPUT = task2h : discrete;
  PROCESSOR = dest1_9, dest2_9, dest3_9,
dest4_9;
  policy = prio;
];

/*TRANS*/ task2h[
  SPORADIC = 0;
  FUNCTION = fun2();
  START_TIME = 1;
  PRIORITY = dest_priority;
  INPUT = store1h : discrete;
  PROCESSOR = dest1_9, dest2_9, dest3_9,
dest4_9;
  ACTIVE = true;
];
/*TRANS*/ task1i[
  PERIOD = per1;
  FUNCTION = fun1();
```

```
  START_TIME = 1;
  PRIORITY = source_priority;
  OUTPUT = store1i : discrete;
  PROCESSOR = source1, source2, source3,
source4;
  ACTIVE = true;
  DEADLINE = dead1;
];

/*STORE*/ store1i[
  TYPE = depletable;
  ELEMENT_SIZE = 10;
  INPUT = task1i : discrete;
  OUTPUT = task2i : discrete;
  PROCESSOR = dest1_10, dest2_10,
dest3_10, dest4_10;
  policy = prio;
];

/*TRANS*/ task2i[
  SPORADIC = 0;
  FUNCTION = fun2();
  START_TIME = 1;
  PRIORITY = dest_priority;
  INPUT = store1i : discrete;
  PROCESSOR = dest1_10, dest2_10,
dest3_10, dest4_10;
  ACTIVE = true;
];
```

APPENDIX I

WORKLOAD (4, 1, 10, 10): FUNCTION FILE

```
FUNCTIONS

EXTERNS
OPER sread;
OPER swrite;

CONSTANTS

CODE

fun1 {
OUTPUT = y: DISCRETE;
BEGIN
LOOP loopcount
{
  swrite(y, WAIT, 0, 200);
};
END;
};

fun2 {
INPUT = x : DISCRETE;
BEGIN
LOOP forever
{
  sread(x, WAIT, NOLIMIT);
};
END;
};
```

APPENDIX J

WORKLOAD $(4, 1, 10, 10)$: EXPERIMENT FILE

EXPERIMENT

CONSTANTS

/**CONSTANT**/ Runs = 1;

PARAMETERS

/**PROCESSOR**/ **default**[
  TIMEUNIT = 210;
  TIMING = true;
  TIMELIMIT = 1000/(loopcount*num_tasks) * per1;
];

BIBLIOGRAPHY

[1] A. K. Agrawala, J. M. Mohr, and R. M. Bryant, "An approach to the workload characterization problem," *IEEE Computer*, vol. 9, no. 6, pp. 18–32, June 1976.

[2] Ö. Babaoğlu, "Efficient generation of memory reference strings based on the LRU stack model of program behaviour," in *Performance '81*, F. J. Kylstra, editor, pp. 373–383, New York, 1981, North-Holland.

[3] Ö. Babaoğlu, "On constructing synthetic programs for virtual memory environments," in *Experimental Computer Performance Evaluation*, D. Ferrari and M. Spadoni, editors, pp. 195–204, New York, 1981, North-Holland.

[4] R. Baird, "APET - a versatile tool for estimating computer application performance," *Software-Practice and Experience*, vol. 3, pp. 385–395, 1973.

[5] W. H. Beyer, editor, *CRC Handbook of Tables for Probability and Statistics*, CRC Press, Inc., second edition, 1987.

[6] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Computer Systems*, vol. 2, no. 1, pp. 39–59, February 1984.

[7] R. Blumofe and A. Hecht, "Executing real-time structured analysis specifications," *ACM Software Engineering Notes*, vol. 13, no. 3, pp. 32–40, July 1988.

[8] R. R. Bodnarchuk and R. B. Bunt, "A synthetic workload model for a distributed file server," *Performance Evaluation Review*, vol. 19, no. 1, pp. 50–59, May 1991.

[9] W. Bruyn, R. Jensen, D. Keskar, and P. Ward, "ESML: An extended systems modeling language based on the data flow diagram," *ACM Software Engineering Notes*, vol. 13, no. 1, pp. 58–67, 1988.

[10] W. Buchholz, "A synthetic job for measuring system performance," *IBM Systems Journal*, vol. 8, no. 4, pp. 309–318, 1969.

[11] M. Calzarossa, M. Italiani, and G. Serazzi, "A workload model representative of static and dynamic characteristics," *Acta Informatica*, vol. 23, no. 3, pp. 255–266, June 1986.

[12] M.-S. Chen, K. G. Shin, and D. D. Kandlur, "Addressing, routing and broadcasting in hexagonal mesh multiprocessors," *IEEE Trans. Computers*, vol. 39, no. 1, pp. 10–18, January 1990.

[13] R. M. Cohen, "Formal specifications for real-time systems," in *Proc. Texas Conference on Computing Systems*, pp. 1.1–1.8. IEEE, 1978.

[14] *ENP K1 Kernel Software User's Guide*, Communication Machinery Corp., May 1986.

[15] M. S. Deutsch, "Focusing real-time systems analysis on user operations," *IEEE Software*, pp. 39–50, September 1988.

[16] P. S. Dodd and C. V. Ravishankar, "Monitoring and debugging distributed real-time programs," *Software-Practice and Experience*, 1992. to appear.

[17] J. W. Dolter, P. Ramanathan, and K. G. Shin, "A VLSI architecture for dynamic routing in HARTS," Technical Report CRL-TR-4-88, Computing Research Laboratory, The University of Michigan, April 1988.

[18] B. Domanski, "Building IMS synthetic workloads," *Perf. Eval. Review*, vol. 13, no. 3 & 4, pp. 23–31, November 1985.

[19] J. J. Dujmovic, "Computer selection and criteria for computer performance evaluation," *International Journal of Computer and Information Sciences*, vol. 9, no. 6, pp. 435–458, 1980.

[20] H. Falk, "CASE tools emerge to handle real-time systems," *Computer Design*, vol. 27, no. 1, pp. 53–74, January 1988.

[21] F. Feather, *Validation of a Fault-Tolerant Multiprocessor: Baseline Experiments and Workload Implementation*, Master's thesis, ECE Dept., Carnegie-Mellon University, Pittsburgh, 1984.

[22] F. Feather, D. Siewiorek, and Z. Segall, "Validation of a fault-tolerant multiprocessor: Synthetic workload implementation," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 303–312, May 1986.

[23] D. Ferrari, "Workload characterization and selection in computer performance measurement," *IEEE Computer*, vol. 5, no. 4, pp. 18–24, July 1972.

[24] D. Ferrari, *Computer Systems Performance Evaluation*, Prentice-Hall, Englewood Cliffs, 1978.

[25] D. Ferrari, "Characterization and reproduction of the referencing dynamics of programs," in *Performance '81*, F. J. Kylstra, editor, pp. 363–372, November 1981.

[26] D. Ferrari, "A performance-oriented procedure for modeling interactive workloads," in *Experimental Computer Performance Evaluation*, D. Ferrari and M. Spadoni, editors, pp. 57–78, New York, 1981, North-Holland.

[27] D. Ferrari, "On the foundations of artificial workload design," in *Proc. of 1984 ACM SIGMETRICS Conf. on Meas. and Modeling of Comp. Sys.*, pp. 8–14, August 1984.

[28] D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems*, Prentice-Hall, Englewood Cliffs, 1983.

[29] S. L. Gaede, "Tools for research in computer workload characterization and modeling," in *Experimental Computer Performance Evaluation*, D. Ferrari and M. Spandoni, editors, pp. 235–247, New York, 1981, North-Holland.

[30] H. Gomaa, "A software design method for real-time systems," *Communications of the ACM*, vol. 27, no. 9, pp. 938–949, September 1984.

[31] H. Gomaa, "Software development of real-time systems," *Communications of the ACM*, vol. 29, no. 7, pp. 657–668, July 1986.

[32] G. Haring, R. Posch, C. Leonhardt, and G. Gell, "The use of a synthetic jobstream in performance evaluation," *The Computer Journal*, vol. 22, no. 2, pp. 209–219, May 1979.

[33] D. J. Hatley and I. A. Pribhai, *Strategies for Real-Time System Specification*, Dorset House Publishing, New York, 1987.

[34] R. Jain, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, Inc., 1991.

[35] E. O. Joslin, "Application benchmarks: The key to meaningful computer evaluations," in *Proceedings of the 20th National Conference*, pp. 27–37. ACM, August 1965.

[36] E. O. Joslin and J. J. Aiken, "The validity of basing computer selections on benchmark results," *Computers and Automation*, vol. 15, no. 1, pp. 22–23, January 1966.

[37] N. I. Kamenoff and N. H. Weiderman, "Hartstone distributed benchmark: Requirements and definitions," in *Proc. Real-Time Systems Symposium*, pp. 199–208. IEEE, IEEE Computer Society Press, December 1991.

[38] D. D. Kandlur, D. L. Kiskis, and K. G. Shin, "HARTOS: A distributed real-time operating system," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 3, pp. 72–89, July 1989.

[39] D. D. Kandlur, *Networking in Distributed Real-Time Systems*, PhD thesis, University of Michigan, 1991.

[40] R. P. Kar and K. Porter, "Rhealstone: A real-time benchmarking proposal," *Dr. Dobb's Journal*, February 1989.

[41] H. F. Ledgard and M. Marcotty, "A genealogy of control structures," *Communications of the ACM*, vol. 18, no. 11, pp. 629–639, November 1975.

[42] D. W. Leinbaugh, "Guaranteed response times in a hard real-time environment," *IEEE Trans. Software Engineering*, vol. SE-6, no. 1, pp. 85–91, January 1980.

[43] J. W. S. Liu. personal communication, May 1991.

[44] C. D. Locke, "Generic avionic software," IBM Systems Integration Division, DRAFT, October 1988.

[45] H. C. Lucas, Jr., "Synthetic program specifications for performance evaluation," in *Proc. ACM Annual Conference*, pp. 1041–1058, Boston, August 1972.

[46] G. A. Ludgate, B. Haley, L. Lee, and Y. N. Miles, "The use of structured analysis and design in the engineering of the TRIUMF data acquisition and analysis system," *IEEE Trans. Nuclear Science*, vol. NS-34, no. 1, pp. 157–161, February 1987.

[47] W. lun Kao and R. K. Iyer, "A user-oriented synthetic workload generator," in *Proc. 12th Int. Conf. on Distributed Computer Systems*, pp. 270–277. IEEE, June 1992.

[48] Luqi, V. Berzins, and R. T. Yeh, "A prototyping language for real-time software," *IEEE Trans. Software Engineering*, vol. 14, no. 10, pp. 1409–1423, October 1988.

[49] H. G. Mendelbaum and D. Finkelman, "CASDA: Synthesized graphic design of real-time systems," *IEEE Computer Graphics and Applications*, vol. 9, no. 1, pp. 40–46, January 1989.

[50] A. K. Mok, "The design of real-time programming systems based on process models," *Proc. Real-Time Systems Symposium*, pp. 5–17, December 1984.

[51] J. Molini, S. Maimon, and P. Watson, "Real time distributed system studies/scenarios," in *ONR Third Annual Workshop: Foundations of Real-Time Computing*, pp. 187–209, October 1990.

[52] A. H. Muntz and R. W. Lichota, "A requirements specification method for adaptive real-time systems," in *Proc. Real-Time Systems Symposium*, pp. 264–273. IEEE, December 1991.

[53] P. W. Oman, "CASE analysis and design tools," *IEEE Software*, vol. 7, no. 3, pp. 37–43, May 1990.

[54] P. Pulli, J. Dahler, H.-P. Gisiger, and A. Kundig, "Execution of Ward's transformation schema on the graphic specification and prototyping tool SPECS," in *COMPEURO-88 System Design: Concepts, Methods and Tools*, pp. 16–25, April 1988.

[55] P. J. Pulli, "Execution of structured analysis specifications with an object oriented petri net approach," in *Proceedings of the IEEE 1989 National Aerospace and Electronics Conference NAECON 1989*, volume 1, pp. 1747–1752, May 1988.

[56] E. L. Reilly and J. W. Brackett, "An experimental system for executing real-time structured analysis models," in *Proceedings, Twelfth Structured Methods Conference*, pp. 301–314, August 1987.

[57] H. D. Schwetman and J. C. Brown, "An experimental study of computer system performance," in *Proc. ACM Annual Conference*, pp. 693–703, 1972.

[58] G. Serazzi, editor, *Workload Characterization of Computer Systems and Computer Networks*, North-Holland, 1985.

[59] A. Singh, *Pegasus: A Controllable, Interactive, Workload Generator for Multiprocessors*, Master's thesis, Carnegie Mellon University, December 1981.

[60] A. Singh and Z. Segall, "Synthetic workload generation for experimentation with multiprocessors," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 778–785, 1982.

[61] *pSOS User's Guide*, Software Components Group, 1986.

[62] K. Sreenivasan and A. J. Kleinman, "On the construction of a representative synthetic workload," *Communications of the ACM*, vol. 17, no. 3, pp. 127–133, March 1974.

[63] H. Tokuda and M. Kotera, "Scheduler 1-2-3: An interactive schedulability analyzer for real-time systems," in *Proc. of the 12th Annual Int'l Computer Software & Applications Conference*, pp. 211–219, 1988.

[64] R. E. Walters, "Benchmark techniques: a constructive approach," *The Computer Journal*, vol. 19, no. 1, pp. 50–55, February 1976.

[65] P. T. Ward and S. J. Mellor, *Structured Development for Real-Time Systems*, volume 1-3, Yourdon Press, Englewood Cliffs, 1986.

[66] M. Webb and P. T. Ward, "Executable data flow diagrams: an experimental implementation," *Structured Development Forum VIII*, August 1986.

[67] J. W. Wendorf, "Implementation and evaluation of a time-driven scheduling processor," in *Proc. Real-Time Systems Symposium*, pp. 172–180. IEEE, December 1988.

[68] S. M. White and J. Z. Lavi, "Embedded computer system requirements workshop," *IEEE Computer*, vol. 18, no. 4, pp. 67–70, April 1985.

[69] B. E. Withers, D. C. Rich, D. S. Lowman, and R. C. Buckland, "Software requirements: Guidance and control software development specification," NASA Contractor Report 182058, Research Triangle Institute, June 1990.

[70] D. C. Wood and E. H. Forman, "Throughput measurement using a synthetic job stream," in *AFIPS Fall Joint Computer Conference*, volume 39, pp. 51–55, November 1971.

[71] M. H. Woodbury, *Workload Characterization of Real-Time Computing Systems*, PhD thesis, The University of Michigan, Ann Arbor, MI 48109, August 1988.