NUREG/CR-6083
UCRL-ID-114565

Reviewing Real-Time Performance of Nuclear Reactor Safety Systems

Prepared by
G. G. Preckshot

Lawrence Livermore National Laboratory

Prepared for
U.S. Nuclear Regulatory Commission

AVAILABILITY NOTICE

Availability of Reference Materials Cited in NRC Publications

Most documents cited in NRC publications will be available from one of the following sources:

1. The NRC Public Document Room, 2120 L Street, NW, Lower Level, Washington, DC 20555-0001

2. The Superintendent of Documents, U.S. Government Printing Office, Mail Stop SSOP, Washington, DC 20402-9328

3. The National Technical Information Service, Springfield, VA 22161

Although the listing that follows represents the majority of documents cited in NRC publications, it is not intended to be exhaustive.

Referenced documents available for inspection and copying for a fee from the NRC Public Document Room include NRC correspondence and internal NRC memoranda; NRC Office of Inspection and Enforcement bulletins, circulars, information notices, inspection and investigation notices; Licensee Event Reports; vendor reports and correspondence; Commission papers; and applicant and licensee documents and correspondence.

The following documents in the NUREG series are available for purchase from the GPO Sales Program: formal NRC staff and contractor reports, NRC-sponsored conference proceedings, and NRC booklets and brochures. Also available are Regulatory Guides, NRC regulations in the *Code of Federal Regulations*, and *Nuclear Regulatory Commission Issuances*.

Documents available from the National Technical Information Service include NUREG series reports and technical reports prepared by other federal agencies and reports prepared by the Atomic Energy Commission, forerunner agency to the Nuclear Regulatory Commission.

Documents available from public and special technical libraries include all open literature items, such as books, journal and periodical articles, and transactions. *Federal Register* notices, federal and state legislation, and congressional reports can usually be obtained from these libraries.

Documents such as theses, dissertations, foreign reports and translations, and non-NRC conference proceedings are available for purchase from the organization sponsoring the publication cited.

Single copies of NRC draft reports are available free, to the extent of supply, upon written request to the Office of Information Resources Management, Distribution Section, U.S. Nuclear Regulatory Commission, Washington, DC 20555-0001.

Copies of industry codes and standards used in a substantive manner in the NRC regulatory process are maintained at the NRC Library, 7920 Norfolk Avenue, Bethesda, Maryland, and are available there for reference use by the public. Codes and standards are usually copyrighted and may be purchased from the originating organization or, if they are American National Standards, from the American National Standards Institute, 1430 Broadway, New York, NY 10018.

DISCLAIMER NOTICE

NUREG/CR-6083
UCRL-ID-114565

Reviewing Real-Time Performance of Nuclear Reactor Safety Systems

Manuscript Completed: July 1993
Date Published: August 1993

Prepared by
G. G. Preckshot

Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, CA 94550

Prepared for
Division of Reactor Controls and Human Factors
Office of Nuclear Reactor Regulation
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001
NRC FIN L1867

Abstract

The purpose of this paper is to recommend regulatory guidance for reviewers examining real-time performance of computer-based safety systems used in nuclear power plants. Three areas of guidance are covered in this report. The first area covers how to determine if, when, and what prototypes should be required of developers to make a convincing demonstration that specific problems have been solved or that performance goals have been met. The second area has recommendations for timing analyses that will prove that the real-time system will meet its safety-imposed deadlines. The third area has descriptions of means for assessing expected or actual real-time performance before, during, and after development is completed. To ensure that the delivered real-time software product meets performance goals, the paper recommends certain types of code-execution and communications scheduling. Technical background is provided in the appendix on methods of timing analysis, scheduling real-time computations, prototyping, real-time software development approaches, modeling and measurement, and real-time operating systems.

Contents

Contents—Appendix A: Real-Time Performance

Contents—Appendix B: Real-Time Systems Complexity and Scalability

Executive Summary

The purpose of this paper is to recommend regulatory guidance for reviewers examining real-time performance of computer-based safety systems used in nuclear power plants. There are three areas of guidance contained in this report. The first area covers how to determine if, when, and what prototypes should be required of developers to make a convincing demonstration that specific problems have been solved or that performance goals have been met. The second area has recommendations for timing analyses that will prove that the real-time system will meet its safety-imposed deadlines. The third area has descriptions of means for assessing expected or actual real-time performance before, during, and after development is completed.

Recommendations for prototype testing suggest that three kinds of prototypes will be useful either for developing or reviewing nuclear reactor safety software. These are human factors prototypes, early prototypes to resolve technical questions, and validation prototypes for performance estimation of the finished software product. Performance models are recommended as a necessary guide for determining the scale of the final prototype.

Recommendations for timing analyses suggest that, with certain restrictions, it is possible to predict lower and upper bounds for code execution time. This ability is a prerequisite for several known methods of producing predictable real-time systems. If the restrictions are not met, it is recommended that the demonstration of predictability be at least as convincing as the proof for an equivalent restricted system.

To ensure that the delivered real-time software product meets performance goals, the paper recommends certain types of code-execution and communications scheduling that can be demonstrated to be correct and to meet deadlines. If these types of scheduling are not used, then it is suggested that proofs be at least as rigorous as the recommended methods. The final recommendation is that performance models and measured performance should converge, and that prototype testing should demonstrate that system performance scales as indicated by the models.

Technical background is provided in the appendix on methods of timing analysis, scheduling real-time computations, prototyping, real-time software development approaches, modeling and measurement, and real-time operating systems. Pointers in the recommendations direct the reader to appropriate locations in the appendices for pertinent information.

Reviewing Real-Time Performance of Nuclear Reactor Safety Systems

1. Introduction

1.1 Purpose

The purpose of this paper is to recommend regulatory guidance for reviewing performance of real-time computer systems used in nuclear power plant safety systems. The focus is threefold: first, to provide guidance on prototyping; second, to recommend timing analyses; and third, to recommend methods of assessing expected or actual delivered performance. The paper is organized specifically to be directly useful to a competent reviewer who does not have special expertise in real-time systems. Two appendices are included for those reviewers wishing more detailed technical justification for the recommendations.

1.2 Scope

This guidance is limited to predictable or partially predictable (a critical-task subset is guaranteed to meet deadlines) real-time systems. Predictability as used here means that there are convincing arguments or mathematical methods that demonstrate that the system will meet its deadlines in all realistic situations. It is assumed that any data communication systems associated with systems considered here are deterministic.

Systems of bounded indeterminacy (the frequency and duration of uncertain operations are bounded) are probably unavoidable, but are sufficiently predictable under certain restrictions. No real-time systems with unbounded-time computations are within the scope of this guidance. Acceptable systems will mainly be statically scheduled, state-based systems, but dynamically scheduled, event-driven systems are admissible if convincing proofs can be made that deadlines will be met.

Fault tolerant systems, for example redundant systems, are not explicitly covered, even though fault tolerance provisions often have impact on system timing. The position is taken here that fault timing can be accommodated by including fault hypotheses during the design process.

1.3 Recommendations and Guidelines Defined

A recommendation is a suggestion that is important to the safety of the system. A guideline is good engineering practice that should be followed to improve the overall quality, reliability, or comprehensibility of the system.

Recommendations in this document have several formats, although all recommendations have a short discussion describing the reason or situation in which the recommendation should apply, followed by the recommendation itself. The question format has a question preceding the discussion and the question poses an issue that the reviewer should resolve. One or more introductory paragraphs may precede a discussion and recommendation to provide background knowledge needed to understand the subject of the recommendation. Introductory paragraphs usually lead off a section of recommendations in a particular subject area.

Following each recommendation, an italicized pointer to further information in the appendices shows the reader where to find more detailed information. The pointer looks like this:

(See Appendix A, Section 2.0, for background on "real-time systems.")

1.4 Structure of This Document

This document is designed to be used by a person who is reviewing a design for one or more real-time computer systems that perform functions that are essential or important to the safety of a nuclear reactor. For this reason, only brief introductory material that is essential for understanding is presented before the several review recommendations. Two appendices that cover the review subjects in more depth are available for reviewers wishing background material. Each recommendation contains a reference to specific sections of the appendix that elucidate the immediate subject of interest.

The recommendations herein cover three areas important to real-time system reviewers: prototyping, timing analyses, and performance assessment.

The first area, prototyping, considers if, when, and what kind of prototypes to require. An important issue—how to determine what size of prototype—is also covered.

The second area reviews three types of timing analysis that can be performed to show that components of a real-time system meet or will meet timing constraints. Timing analysis is necessary, but insufficient, to demonstrate that a system is predictable.

The last area recommends methods for assessing real-time performance. Performance assessment is not a simple matter of trying a few representative input sequences, but must be guided by simulation and modeling. The recommendations discuss the simulation, modeling, and performance testing required to gain confidence that the system will perform as intended.

1.5 Motivation for This Guidance

Experience in the real-time world in general has resulted in many horror stories of real-time systems that are delivered late, are too slow, and are economically unfixable. The nuclear power industry has its own special horror story, the French Controbloc P20 experience. Among other faults, the P20 portion of a reactor protection system did not perform well enough to protect the reactor and was economically unfixable. The French eventually replaced the Controbloc P20 with a simpler, predictable system.

This guidance places current results in predictable real-time systems at the disposal of the reviewer. These results are not state-of-the-art in terms of getting maximum performance from a real-time computer system, but they are state-of-the-art in conservative, reliable design of predictable real-time systems. To avoid the unpleasant surprises that occurred late in the French experience, the advice given includes techniques that provide reviewers with early warnings of incipient performance deficits.

1.6 Special Knowledge Required

Deadline

A time delay measured from the time an input enters the system until the time a correct output must be produced. Deadlines are central to current real-time system theory, and the performance of a real-time system is specified in terms of deadlines that it must meet. See the entry describing real-time systems for more detail.

Deterministic

A "deterministic" communication system delivers messages within a finite, predictable time delay that is a function of system communication load. Determinism is considered to be a very important quality for communication systems used in real-time applications. Unfortunately, no communication system is deterministic under all conditions, and unpredictable delays will be incurred for at least some errors or failures. The term "deterministic" can be applied to systems or hardware other than communication systems and it means the same thing: finite, predictable time delay that is a function of a load parameter.

Dynamic schedule

A schedule for when to run programs or subroutines that is calculated during system execution. To demonstrate that a dynamically scheduled real-time system will meet its deadlines, it must be shown that the scheduling algorithm will always schedule routines with sufficient computational time. This is not an easy thing to do without some restrictions.

Performance model

A mathematical model that includes interconnections (between programs, hardware, or combinations of the two), program or subroutine execution times, interconnect delays, and sometimes resource dependencies. Performance models are used to predict system time response to inputs. Performance models can treat execution times and delays statistically or exactly, and the results obtained will be statistical distributions of response times or exact low and high limits. The two mathematical modeling tools used most often to make performance models are Petri nets and queuing models. An often unrecognized performance model is the static schedule. Performance models are sometimes executed by digital computers, in which case they are also known as simulations.

Petri net

An abstract, formal model of information flow, showing static and dynamic properties of a system. A Petri net is usually represented as a graph having two types of nodes (called places and transitions) connected by arcs, and markings (called tokens) indicating dynamic properties. (See IEEE Std 610.12-1990.) Petri nets are often used to make performance models.

Queuing network model

An abstract, formal model of job flow, showing statistical timing properties of a system. Programs, routines, or communication links are modeled as service providers at the head of queues, and each service has a statistical distribution of service times. The output of such models is "throughput," or the average amount of work the modeled system can be expected to perform.

Real-time system

Real-time systems are digital systems which must produce correct output in response to input within well-defined deadlines.

1. Real-time systems have inputs connected to real-world events.
2. Events (inputs) cause computations to occur that finish before well-defined deadlines, measured from the event.
3. Computations produce outputs that are connected to the real world.

Software for real-time systems has an additional correctness constraint: even if the values produced by software are correct, the software is considered to fail if it does not meet its deadline. Current-day practice further divides real-time systems into *hard real-time systems*, and *soft real-time systems*. Hard real-time systems cannot tolerate any failures to meet deadlines. Soft real-time systems can tolerate some deadline failures and will still function correctly.

Static schedule

A schedule for when to run programs or subroutines that is calculated prior to system build. In other words, the schedule is a fixed rota that is repeated indefinitely. To make a static schedule, the scheduler must know the execution time limits and communication path delays of all programs and communication paths that it will schedule. The advantage of such a schedule is that it can be proved prior to system installation that the system will meet its deadlines, and it is not necessary to show that a scheduling algorithm will work under all circumstances. A static schedule is a de facto performance model because system performance can be directly determined from the schedule.

2. Guidance

2.1 Prototype Testing

The issue in prototype testing is determining if, when, and what kinds of prototypes are required to demonstrate sufficient system performance so that a safety-critical, real-time system will not fail because of inadequate real-time performance.

2.1.1 Human Factors

Human interaction with nuclear reactor safety systems occurs during surveillance, maintenance, and display of safety parameters and trip status. Many other related human factors issues are involved, but slow real-time system response to operator actions has been implicated in many operator errors. Although reactor control systems are non-safety systems, slow response of a reactor control system or process parameter display system may induce an operator to take actions that challenge the protective system.

Recommendation: Human factors analysis of operator interactions with the reactor protective system should be performed, and, if needed, a prototype with nominal response times should be used to answer any outstanding questions resulting from the analysis.

Guideline: Human factors prototypes are standard engineering practice for advanced control systems. A human factors prototype of the reactor control system is suggested to help ensure that inadvertent operator actions do not challenge the reactor protective system.

(See Appendix A, Section 3.4, for more background on prototyping.)

2.1.2 Non-Deterministic Hardware

The execution speed of some hardware and communication gear is variable within fixed limits because of (but not limited to) interrupt latency, memory refresh interference, and media access methods. This bounded indeterminacy can introduce inaccuracy into performance models if it is not accounted for.

Recommendation: An early prototype with test software or some other equivalent method should be used to measure computer system and communication gear indeterminacies that will be present in the delivered product. The measured figures should be used as inputs to performance models.

(See Appendix A, Section 3.4, for more background on prototyping. See Section 3.2 for a discussion of bounded indeterminacy.)

2.1.3 Scale of Engineering Prototype

It is very difficult *a priori* to decide how much of an unknown design should be constructed to demonstrate that the final system will perform as planned. Refined performance models and analysis of the design architecture, including communication systems, are often necessary to determine how system performance will scale in the final configuration.

Recommendation: The extent of the system performance demonstration prototype should not be decided until the system architecture has been analyzed and refined performance models are available to predict performance scaling.

(See Appendix A, Section 3.6, for details on modeling. See Appendix B, Section 3, on Scalability.)

2.1.4 Demonstrating Prototype Correctness

At the time an engineering prototype is constructed, it may be late in the software development process, or the prototype may have undergone modifications to correct discovered problems. The assumptions that early performance models are based upon may therefore be false.

Recommendation: Prior to using an engineering prototype for system performance tests, the prototype should be analyzed and tested to ensure that it embodies the assumptions used to make current performance models, and that the prototype accurately reflects the delivered system.

(See Appendix A, Section 3.6, for details on measurement.)

2.1.5 Engineering Prototype Performance

Without an understanding of how a distributed system achieves its performance, it is difficult to devise tests that stress the system or that demonstrate performance scaling. Randomly selected input sequences may also be used (Zucconi and Thomas 1992) but models may be required to choose feasible random input trajectories (connected subsequences of inputs).

Recommendation: Test suites to demonstrate absolute performance, performance scaling, overload performance, and response to randomly selected input sequences should be designed using refined performance models for guidance. All unpredicted prototype responses should be investigated and resolved.

(See Appendix A, Section 3.6, for details on measurement. See Zucconi and Thomas (1992) for more extensive details on testing software.)

2.2 Timing Analyses

Simple timing analyses are necessary but not sufficient to ensure that a real-time system will meet its deadlines. The three kinds of timing analysis considered in this paper are module timing analysis, system timing analysis, and schedulability and priority analysis.

2.2.1 Predictable Software

With the current state of the art it is necessary but not sufficient to have predictable bounds on the execution time of software that runs in a system, in order to be able to predict that the system will meet its deadlines. Ways to achieve this include hard limits for code modules, imprecise calculations, hard limits for groups of code modules, or aborts if deadlines are passed.

Recommendation: Software that is used in safety-critical systems should have deterministic execution time bounds. One of the above-named methods or other means should be used to ensure that safety-critical code execution times are predictable. The demonstration should include measured effects of hardware indeterminacy.

(See Appendix A, Section 3.1, for a discussion of ways to make predictable real-time systems. See Section 3.2 on timing and deterministic execution time bounds.)

2.2.2 Timing Analysis of Modules

Mechanized timing analysis of code modules from which certain coding practices[1] have been excluded has been shown to be possible. Preemption, exceptions, and resource blocking are not accounted for by simple code module analysis, and are excluded except when bounded indeterminacy can be proved. An acceptable method of demonstrating module predictability is to restrict coding practices so that mechanized timing analyzers can predict code lower and upper execution time bounds from examination of source code, allowing for bounded indeterminacy. This is called static timing analysis.

Recommendation: Static timing analysis or an analysis of equivalent rigor should be performed and documented on all modules of safety-critical software.

(See Appendix A, Section 3.2, on timing analyses.)

2.2.3 Risky Practices

In real-time safety systems that allow preemption, exceptions (including interrupts), resource blocking, or coding practices that cause timing analysis of source code to be indeterminate, simple module timing analysis is either insufficient or impossible.

Recommendation: The methods used to demonstrate predictability of real-time safety systems using risky practices should be as convincing as static timing analysis. Making such a convincing

[1] For an example, see Appendix A, Section 3.2, Timing. Specifically, see the limitations imposed by the Mars development environment.

demonstration has proved to be difficult for unrestricted software, which is why some software practices are regarded as risky.

(See Appendix A, Section 3.2, on things that make timing analysis uncertain, and restrictions that reduce analysis problems. See Section 3.3 on resource blocking and scheduling. See Appendix B, Section 2.1, on System Complexity, and Section 2.1.3 on Complexity Attributes.)

2.3 Assessing Real-Time Performance

Real-time performance requirements are assumed to be derived from system safety requirements, and are assumed to be a set of deadlines, display update times, and data throughput minimums for the performance of listed safety functions. Additional subsidiary performance requirements may be derived from the core requirements, depending upon system architecture. The entire set of requirements is known as the *real-time system performance requirements*.

2.3.1 Scheduling and Deadlines

Statically scheduled real-time systems have been shown to be predictable if execution times of individual code modules are predictable. Certain kinds of dynamically scheduled systems are also predictable, in that a subset of critical tasks can be shown to meet deadlines.

Recommendation: The method used to demonstrate that all critical tasks of a safety-critical real-time system will meet their deadlines should be as convincing as an equivalent statically scheduled system.

(See Appendix A, Section 3.3, the first two paragraphs on the benefits of static scheduling. Dynamic scheduling methods are discussed later in the Section. See the bullets at the end of the section for a reprise of important points about scheduling.)

2.3.2 Communication Scheduling

Even though a communication system is deterministic, overall system deadline performance may still fail if insufficient time is allowed for message transmission, or critical messages conflict in time.

Recommendation: Communication system usage should be included in scheduling. Demonstration that message arrival is timely should be as convincing as an equivalent statically scheduled system.

(See Appendix A, Section 3.3, on scheduling, and Sections 3.5.2 and 3.7 for a description of how an example real-time development system handles communication scheduling.)

2.3.3 Model Validation

One serious difficulty with models is that they can diverge from reality if not checked against an example of the thing they model. Model validation tests are different from system performance tests because model validation tests specifically explore features and limits of the model. Predictions taken from a divergent model are spurious.

Recommendation: Safety-critical performance models should be validated by tests on prototype (or production) hardware that has been checked to ensure that hardware assumptions underlying the model are correct.

(See Appendix A, Section 3.4, on prototyping and Section 3.6 on modeling and measurement.)

2.3.4 Model and Performance Convergence

Performance of a delivered system configuration is usually extrapolated from performance tests on prototype or reduced sets of hardware. Extrapolation is done using a performance model (to be preferred) or informal "engineering judgment" (to be avoided if possible). A performance model can provide guidance for devising performance tests of the delivered system and predictions of selected performance measures of the delivered configuration.

Recommendation: Performance of the delivered product and predictions of validated performance models should converge. All deviations from expected performance should be explained, and either the delivered system or the model should be corrected. Final performance should meet the detailed real-time system performance requirements.

(See Appendix A, Section 3.4, on prototyping, and Section 3.6 on modeling and measurement. See Appendix B, Section 3, on Scalability.)

2.3.5 Scheduling in Non-Safety Systems

Dynamically scheduled systems are more difficult to predict than statically scheduled systems.

Guideline: It is good practice to use static scheduling for real-time systems unless functional variation is so great that dynamic scheduling cannot be avoided. Even in this case, it may be possible to partition the real-time system so that essential functions can be statically scheduled.

(See Appendix A, Section 3.3, on scheduling, and Section 3.7 on operating systems that use combinations of dynamic and static scheduling.)

2.3.6 Essential Functions in Non-Safety Systems

Non-safety real-time reactor control and process parameter display systems can affect reactor safety by inducing operator actions that challenge the reactor protection system. Non-safety system performance has been implicated in this because late operator feedback may cause incorrect perception of reactor conditions.

Guideline: It is suggested that non-safety real-time computer systems in nuclear reactors be reviewed for functions that may cause challenges to the reactor protection system. These functions may be termed *essential functions,* and steps may be taken to ensure that the essential function subset of reactor control systems or process parameter display systems always meet their deadlines.

Appendix A:

Real-Time Performance

G.G. Preckshot

Manuscript Date: May 28, 1993

Contents

Executive Summary

This paper has three specific goals for regulatory guidance. The first is to determine if, when, and what prototypes should be required of developers to make a convincing demonstration that specific problems have been solved or that performance goals have been met. The second goal is to present and recommend timing analyses that will prove that the real-time system will meet its safety-imposed deadlines. The third is to suggest means for assessing expected or actual real-time performance before, during, and after development is completed.

The thesis of the paper is that safety-critical real-time systems must be predictable in order to satisfy the three goals. In this paper, technical background is provided on methods of timing analysis, scheduling real-time computations, prototyping, real-time software development approaches, modeling and measurement, and real-time operating systems. The background material supports recommendations that guide reviewers and developers in prototype testing, timing analyses, and ensuring that real-time performance requirements are met.

Research results developed for prototype testing suggest that three kinds of prototypes will be useful, either for developing or reviewing nuclear reactor safety software. These are human factors prototypes, early prototypes to resolve technical questions, and validation prototypes for performance estimation of the finished software product. Performance models are useful as a guide for determining the scale of the final prototype.

Work in the literature on timing analyses suggests that, with certain restrictions, it is possible to predict lower and upper bounds for code execution time. This ability is a prerequisite for several known methods of producing predictable real-time systems. If the restrictions are not met, then any demonstration of predictability should be at least as convincing as the proof for an equivalent restricted system.

To ensure that the delivered real-time software product meets performance goals, this paper covers certain types of code-execution and communications scheduling that can be demonstrated to be correct and to meet deadlines. If these types of scheduling are not used, then alternative proofs should be at least as rigorous as the described methods, or doubts will exist that the delivered system will perform under all circumstances.

Finally, current practice in the use of performance models is discussed. Performance models and measured performance should converge, and prototype testing should demonstrate that system performance scales as indicated by the models.

Real-Time Performance

1. Introduction

Real-time computer systems are now used in some nuclear reactor control systems and protection systems. Recent failures (the French Controbloc P20) and validation difficulties (Ontario Hydro's Darlington Plant) have focused attention on system performance and software reliability determination. This paper reviews methods for building real-time computer systems for reactor protection systems to meet performance goals, and offers recommendations for regulators who must determine if appropriate methods were followed during development, whether the system design is likely to meet performance requirements, and what testing or prototype testing should be done to verify this.

1.1 Goals

This paper has three specific goals for regulatory guidance. The first is to determine if, when, and what prototypes should be required of developers to make a convincing demonstration that specific problems have been solved or that performance goals have been met. The second goal is to present and recommend timing analyses that will prove that the real-time system will meet its safety-imposed deadlines. The third is to suggest means for assessing expected or actual real-time performance before, during, and after development is completed.

1.1.1 Need for Prototype Testing

The computer science meaning of the word "prototype" is not only double-faceted but differs from the traditional engineering use of the term. Engineers mean "first of type" or "first off the line," and an engineering prototype is normally used to perform system tests to verify either design correctness or manufacturability. In contrast, software engineers use prototypes to demonstrate system–user interactions to prospective users, or to determine if specific design approaches will work. The former software engineering use is to refine user requirements, while the latter is to demonstrate that there are feasible ways to meet the requirements.

Of the three kinds of prototypes, software engineering feasibility prototypes and traditional engineering prototypes are most applicable to real-time systems for nuclear reactor protection systems. This will be discussed at more length in Section 3.4.

1.1.2 Timing Analyses

Timing analyses are performed to reduce the probability that expected or unexpected combinations of data values or event sequences will cause real-time systems to miss deadlines. A deadline, in this context, is a time limit on a real-time system's response to specific stimuli. If timing analyses are not

performed, then real-time software remains a black box whose deadline performance in all situations is uncertain.

There are several kinds of timing analysis, of which three types will be discussed in this report. The simplest, and most obvious, is computing expected code run time by adding up individual instruction execution times. This method works well for code modules, but does not address the timing of module interactions, which is the second analysis method addressed. The third kind of analysis, intermodule sequencing, is not directed at absolute timing at all, but at whether code modules execute in correct order of precedence. The three methods can be considered, respectively, as module timing analysis, system timing analysis, and schedulability and priority analysis.

1.1.3 Assessing Real-Time Performance

A naíve approach to assessing real-time system performance suggests that the completed system should be tested under a selection of work loads, and performance should be deduced in the manner of final acceptance testing of electrical machinery. Unfortunately, the design of distributed, interconnected computer systems admits far more complexity and variability than traditional electric machine design, so that performance assessment is a process that begins in the initial system design stages. The difficulty is that there is no commonly accepted system model as there is with electrical machinery. Consequently, unless a specifically tailored system model is developed while the system is being designed, it is very difficult to devise performance tests and work loads that adequately measure realistic system performance.

Two general kinds of system model are popular—queuing models and Petri net models. Each has been enhanced by various investigators in attempts to compensate for real deficiencies in the models, with the result that a possibly confusing number of subtypes now exist. Each model benefits from data produced by timing analyses, and is in turn used in later stages of system timing analysis for simulation and analytic modeling. The usual progression, which will be elaborated later, is initial modeling to project expected system performance, model refinement and use for development guidance during intermediate development stages, and final resolution of conflicts and misunderstandings during performance testing and system validation. Ideally, at the end of the process, the reviewer, the developer, the model, and the system produced will all agree.

1.2 Structure of This Paper

The balance of this paper addresses the above goals. However, before discussing the scope of the paper's coverage (Section 2.4), a number of general issues about real-time systems, including definition, are introduced to make succeeding comments about scope intelligible. After delineating scope, a technical background section introduces predictability, timing, and scheduling, and then considers prototyping, development approaches, and modeling and measurement. A short subsection on a limited set of real-time operating systems is provided to introduce pertinent characteristics of such products. Finally, recommendations and guidelines are provided for prototyping, timing analyses, and assessing real-time performance of candidate systems.

2. Real-Time Systems Defined

Real-time systems are computer systems which must produce correct outputs in response to real-world inputs within well-defined deadlines (Burns and Wellings 1990). This definition places an additional correctness constraint on software for real-time systems: even if the values produced by software are correct, the software is considered to fail if it does not meet its deadline. Current-day practice further divides real-time systems into hard real-time systems and soft real-time systems. Hard real-time systems cannot tolerate any failures to meet deadlines. Soft real-time systems can tolerate some deadline failures and will still function correctly. An example of the former is any real-time aircraft or missile flight control system. A deadline failure in either of these systems could cause loss of aircraft or missile. Airline flight reservation systems are examples of soft real-time systems. An occasional missed deadline simply delays a reservation clerk and irritates customers. Current research in real-time systems has identified a number of important issues and misconceptions. It is popularly assumed that fast computing is real-time computing. This is untrue. While speedy computers undoubtedly help by reducing computation time, real-time systems are concerned with meeting deadlines rather than going as fast as possible (Stankovic 1988). This also accounts for the difference between "performance engineering" and real-time systems, with the former using statistical performance indices that permit variability that the latter cannot tolerate.

A second important theme illustrated by the difference between the conventional performance engineering design approach and real-time system design is predictability. Real-time system developers seek to guarantee timing and schedulability of tasks to ensure that the real-time product meets deadlines with absolute predictability (Natarajan and Zhao 1992). There are two approaches to this problem, and they can be characterized as static versus dynamic scheduling. A statically scheduled real-time system has a schedule or schedules that have been previously calculated, while in a dynamically scheduled system, some or all of the tasks are scheduled at unpredictable times by a real-time operating system or dedicated scheduler. Even though tasks are scheduled at unpredictable times in dynamically scheduled systems, it still may be possible to predict that deadlines will be met for a subset of tasks. This is usually more difficult to do than it would be for a statically scheduled system.

Statically scheduled systems are typically designed for worst-case load, and therefore may use more resources than dynamically scheduled systems. Resources include CPU time, memory, communication link bandwidth, and the use of various devices such as printers, keyboards, displays (CRTs), and the like. The advantage of statically scheduled systems is that, for real-time applications that have a well-known rota of functions to accomplish, it is possible to produce software systems with very convincing guarantees of deadline performance. The relative simplicity of the approach also has merit for safety- or life-critical systems.

Static systems do not do well in applications where tasks are not fixed—for example, systems with unpredictable function demands or unpredictable computations (recursions, convergence, data-dependent algorithms) which may extend execution times beyond planned limits. In these cases, dynamically scheduled systems can provide more functional flexibility and better resource usage, at the expense of more difficult solution of resource requirements, availability, and timing guarantee problems. The hardest problem is demonstrating that dynamically scheduled systems will handle overloads and faults acceptably, since these systems are not designed or sized for worst-case operation.

Both statically and dynamically scheduled systems will be considered in following sections. Scheduling is central to current real-time systems theory, and the subject is interwoven through many of the subjects to be treated.

2.1 Complexity

A significant issue for the reviewer is the complexity of a candidate real-time system and how it will affect the difficulty of ensuring adequate performance. In a previous paper (reprinted in this report as Appendix B) an approximate complexity scale was introduced, and it is instructive to consider that scale in light of what has just been written about predictability and scheduling. For convenience, the complexity attributes are repeated below:

- Timing complexity.
- Priority complexity.
- Resource complexity.
- Connection complexity.
- Concurrency.
- Communication delay.

In general, statically scheduled systems are progressively more difficult to schedule as progressively more complexity attributes are present in the design. However, a static schedule, once made, can be analyzed for correctness and deadline conformance. Since the schedule calculations are made off-line, the computational expense and delay incurred by lengthy scheduling algorithms are not factors in system performance. Developers therefore have considerable latitude in tools used to produce an acceptable task and communication schedule, and more complex system interactions may be provably safe and timely.

In contrast, dynamically scheduled systems exhibit at least timing complexity and, if resources are shared between tasks and preemption is allowed, there is priority and resource complexity as well. Unlike statically scheduled systems, where it is only necessary to prove *a posteriori* that the schedule is correct, here it must be proven *a priori* that an unknown schedule will be correct. Priority and resource interactions considerably complicate this problem. The addition of connection complexity (intertask communication), concurrency, and communication delay may make definitive proofs of correctness and deadline performance impossible without significant restrictions on the software design.

2.2 State-Based Systems

Another important distinction that applies to real-time systems is whether they are state-based or event-driven systems. These are idealizations of two approaches to overall design of real-time systems. Differences between the two have been elaborated by Kopetz (circa 1990) and discussed with regard to communication systems by Preckshot (1993). State-based systems transmit "state" (data that describe external and internal conditions) at regular intervals and recalculate control outputs after each state exchange. Because most state information is unchanging or slowly changing, this method results in inefficient use of communication resources and CPU bandwidth. However, the system is not subject to overload because the same amount of data is transmitted no matter what the circumstances are. The elimination of the overload performance problem is often regarded as ample recompense for less efficient use of resources, and state-based systems are preferred in ultra-reliable applications.

2.3 Event-Driven Systems

Event-driven systems transmit information only when "significant" events occur and recalculate control outputs at event times. Leaving aside how "significant" is defined for the moment, this method

results in less resource usage during quiet periods, but may cause system overload and failure to respond when control or protective action is most needed. Event-driven systems are inappropriate for applications where regular, precisely timed control outputs are required (for example, closed-loop control), but are often used where intermittent, non-linear actions are required (such as contact closures to start or stop motors). Hybrid event-driven and state-based systems are possible.

The question of what is "significant" is sometimes crucial to system operation. Too strict a bound on significance (e.g., change in value required is too large) may result in the system being insensitive to something it should see. Too loose a bound (e.g., small changes in value are "significant") may result in frequent overloads and degraded performance. In general, the performance of event-driven systems is more difficult to predict than state-based systems (Kopetz circa 1990). Therefore, some event-driven or hybrid systems may be unacceptable for safety applications because a final safety determination is too difficult or is impossible to make.

2.4 Scope

Reviewers of safety-critical software are unlikely to be satisfied with unpredictable real-time systems used in safety-critical systems, and are not pleased to encounter the same in important non-safety systems such as reactor control systems. Except for examples needed to illustrate a point, this document is limited to predictable or partially predictable (a critical-task subset is guaranteed to meet deadlines) real-time systems. Unpredictable real-time systems impose such a burden of proof during safety determination that they are effectively ruled out in any safety system where consequences of failure are "unacceptable." Predictability as used here means that there are convincing arguments or mathematical methods that demonstrate that the system will meet its deadlines in all realistic situations. It is also assumed that any communication systems associated with systems considered here are determinate (see Preckshot 1993).

As will be developed in the section on timing (Section 3.2), systems of bounded indeterminacy (the frequency and duration of uncertain operations are bounded) are probably unavoidable, but are sufficiently predictable under certain restrictions. No real-time systems with unbounded-time computations are within the scope of this report. The report discusses mainly statically scheduled, state-based systems, with the proviso that dynamically scheduled, event-driven systems are admissible if convincing proofs of similar rigor can be made that deadlines will be met.

Fault-tolerant systems—for example redundant systems (Lala et al 1991)—are not covered, even though fault tolerance provisions often have impact on system timing. The position is taken here that fault timing can be accommodated by including fault hypotheses during the design process.

3. Technical Background

Guidelines for reviewing safety-critical real-time systems are not credible without technical support. This section provides the technical basis for the review guidelines proposed in Section 4. Since predictability is central to the thesis of this paper, some known methods of achieving it are discussed. Two subjects basic to deadline performance prediction, timing analysis and scheduling, are covered next. Then three areas important for implementing and testing real-time systems (prototyping, development approaches, and modeling and measurement) are covered. Lastly, there is a limited discussion of real-time operating systems that introduces some important features.

3.1 Predictable Real-Time Systems

Rather than try to answer the difficult general question, "What makes a system predictable?" this paper addresses the easier question, "What approaches have produced predictable systems?" There may be other ways of producing predictable real-time systems than those described here, but within the current art there is no general solution to the problem.

One way to demonstrate that a system will meet its deadlines is to show two things: that all tasks are schedulable, and that all tasks will be completed successfully before their deadlines if executed on schedule. A set of tasks is schedulable if, given that resource and priority conflicts are resolved, each task is completed prior to the time that it is scheduled for another execution, and CPU utilization (ratio of active CPU time to active plus idle time) is less than 1.0. For a dynamically scheduled system with unpredictable task phasing (task start times with relation to each other) and cycle times (period of task repetition), schedulability is a difficult thing to prove in general. In statically scheduled systems that do not stop, elementary reasoning suggests that tasks execute cyclically after an initial startup period (because the schedule is finite, but system operation time is unbounded). For such systems it can be shown that an independent task set (Ω: $\tau_i \in \Omega$) is schedulable by the rate-monotonic scheduling algorithm[2] if

$$C_1/T_1 + C_2/T_2 \ldots + C_n/T_n \leq U(n) = n(2^{1/n} - 1) \leq 1$$

where

n is the number of tasks

C_i is the execution time for task τ_i

T_i is the period time for task τ_i

U(n) is the least-upper-bound CPU utilization for n tasks.

(Sha and Goodenough 1990; Liu and Layland 1973)

The assumptions behind this expression are that task execution times are known, period times are known, and that several different period times (causing tasks to periodically vary phase between one another) can be accommodated in the same schedule. In other words, tasks are not required to have the same period. This still does not ensure that a non-conflicting schedule of dependent tasks can be attained (see rate monotonic scheduling in Section 3.3), but does indicate that task timing bounds and period times are a necessary condition for schedulability, and sufficient to guarantee schedulability of independent tasks. The rate monotonic schedulability relation above is optimal in the sense that no other fixed-priority scheduling algorithm can schedule a task set that the rate monotonic algorithm fails to schedule (Liu and Layland 1973).

If the deadline for task τ_i is D_i, then the deadline constraint offers two further bits of information. First, $D_i < T_i$, and second, $C_i < D_i$, where it is assumed that D_i is measured relative to the start of the task cycle.

For systems that do not directly satisfy the assumptions made above, certain regularity constraints may still allow predictability. Kopetz et al (1991) characterize hard-real-time systems based on real-time transactions, by which is meant a connected stimulus–calculation–response string of events. Such systems can be regularized if they can be restricted by a load hypothesis and a fault hypothesis. The load hypothesis defines a finite number of kinds of transactions, a non-zero minimum time between transactions of the same kind, and a non-zero minimum time between system recognition of stimuli. The fault hypothesis makes the same regular definitions of faults to which the system will respond. If the

[2] A preemptive scheduling algorithm in which task priority is inversely proportional to task period.

maximum computational time for each kind of transaction can be bounded, it can be shown that systems under regularity constraints have schedulability and deadline performance bounded by cyclic, rate-monotonic, fixed-priority systems of appropriately selected parameters.

Some dynamically scheduled systems can be made predictable for a subset of critical tasks by imposing regularity constraints that permit the reduced task set to meet the schedulability criteria above. The critical tasks must be able to preempt non-critical tasks, and the blocking time caused by resource conflicts with non-critical tasks must be included in the schedulability expression above (Sha et al 1990). This will be explained further in Section 3.3, Scheduling.

The criteria for predictability and schedulability as presented here are therefore:

- The system is periodic, and
- Task cycle periods are known, and
- Task execution times are bounded and known, and
- Deadlines are shorter than the task cycle, and
- Task runtime to completion is shorter than the deadline, and
- The task set satisfies the rate-monotonic least-upper-bound (LUB).

or:

- The system satisfies regularity constraints, and
- Bounding static system satisfies the rate-monotonic LUB.

or:

- A critical task subset satisfies regularity constraints, and
- The critical tasks preempt non-critical tasks, and
- Blocking time is included, and
- Bounding static system satisfies the rate-monotonic LUB.

Note that predictability does not imply feasibility. Resource conflicts may still make scheduling impossible.

3.2 Timing

Knowledge of timing behavior is central to predictability of real-time systems. The most basic timing knowledge describes the behavior of individual program units or hardware, and was called "module timing analysis" earlier in this report. This information is necessary to use schedulability expressions and scheduling algorithms, and is the first subject of this section. However, a single execution time is rarely defined for even simple segments of code, so some method for reasoning about execution times must be devised that includes timing variabilities. For hard real-time systems, the stochastic measures used in statistical performance analysis are not precise enough, so a way to speak of hard execution time bounds is needed. A method proposed by Shaw (1989) is discussed to illustrate the important issues. Some practical methods (software tools) for analyzing program unit execution time bounds are discussed, along with restrictions on unit software design. Finally, in distributed systems, separated parts of such systems do not perceive the same times, so that misorderings, anachronisms, and synchronization errors can occur unless measures are taken to prevent them. Some results in this area are introduced.

Hasse (1981) considered the execution of tasks synchronizing through critical sections (shared code that updates shared variables) in an attempt to estimate the time spent during these interactions. This work introduced the idea of "execution graphs," in which alternative sequences of access are enumerated and timing behavior is deduced for feasible orderings. Similar work on single programs used

a "program graph" (Oldehoeft 1983) to enumerate alternative paths and simplify the expressions so obtained. Both of these papers demonstrated methods by which timing could be analyzed for either single or cooperating programs, and which appear to be the basis for later program unit timing analysis tools. Neither paper considered the effects of exceptions in detail.

Woodbury (1986) derived an expression for the probability distribution function for successful task completion time, including exception effects. The derivation includes two parts, one for exception probabilities and one for task time distributions. Restrictions were placed on admissible tasks (later echoed or repeated in part by other workers). These restrictions were:

- Single entry/single exit.
- No preemption.
- Input/output is deterministic and each occurs at most once per task execution.
- Deadlines are met by abort if necessary.[3]

With these restrictions, a regular task structure graph was developed that permitted construction of analyzable real-time task graphs from elementary building blocks. By assigning *a priori* probabilities to branches and exception probabilities to individual instructions, Woodbury deduced probability distribution functions for active task time (F_{act}) and exception occurrence (F_{exc}). Combining these leads to:

$$\text{Prob}[T \le t, \text{task done}] =$$
$$\{1 - F_{exc}(t)\}F_{act}(t) \qquad \text{if } t < t_d$$
$$\{1 - F_{exc}(t_d)\}F_{act}(t_d) \qquad \text{otherwise}$$

where t_d is deadline time.

This work is significant because it demonstrates an analysis method that includes exception probabilities and provides a probabilistic estimate for successful task completion. Two useful results are that the analysis suggests ways to minimize $F_{exc}(t)$ and maximize $F_{act}(t)$. Since predictable systems need virtual certainty about task timing, a probability distribution of task completion times that approaches 1.0 before t_d is quite useful.

While the work of Hasse, Oldehoeft, and Woodbury showed that there were no logical barriers to task timing predictability, one very important factor, hardware, was not dealt with in detail. Park and Shaw (1991) have found that some hardware with dynamic memory refresh may exhibit as much as 7% excess random delay unaccounted for by instruction time addition. In another report (Callison and Shaw 1991), as much as 15% random delay due to memory refresh and bus synchronization during interrupt service was found. This is disturbing but not unmanageable if the indeterminacy is bounded. "Interferences" of the types reported can be handled if the frequency and duration of the interference are bounded. An example given for a clock interrupt illustrates the method. An obvious extension permits calculation of minimum and maximum bounds on execution time.

$$t^m_p = t'_p \left(\frac{p_{clock}}{p_{clock} - t_{clock}} \right)$$

where

t^m_p is the measured (actual) program execution time

t'_p is the pure (calculated) program execution time

[3] A technique mentioned in Section 3.3, Scheduling, called "imprecise computation" replaces failed (overtime) calculations with a quick approximation, allowing deadlines to be met.

p_{clock} is the clock period time
t_{clock} is the clock interrupt service time.

The reports by Shaw and others suggest that no realistic modern computer system is exactly predictable, but that bounds on minimum and maximum task execution times can be calculated. Shaw (1989) proposed a formal method of reasoning about such timing behavior. In precis, since earlier tasks have uncertain ending times, the starting time of a new task τ may be uncertain but in a bounded interval (a,b). Because τ also contains some uncertainty, it finishes in a bounded interval (c,d) that is not necessarily a simple offset of (a,b). Thus, given a sequence Σ of tasks that starts in (p,q), an algebra of intervals is required both to determine the ending interval of Σ or the starting and ending intervals of any $\tau_i \in \Sigma$. Timing tools for real systems must use some variant of this reasoning if accurate results are expected. The utility of lower bounds on task execution times becomes immediately obvious, since both upper and lower bounds are required for interval arithmetic.

Timing analysis tools that use the principles just mentioned have been devised and demonstrate the feasibility of performing the analysis. Park and Shaw (1991) have built a timing analysis tool which analyzes a subset of the C programming language. The tool uses a compiler front-end to generate a syntactical analysis of program structure, which is then characterized in terms of "atomic blocks," or compiler-generated machine code sequences of known timing behavior. The block structure was then combined under time reasoning rules to produce lower and upper execution time bounds for code segments or programs. A limited number of tests were performed for small and large atomic block granularities, with generally successful predictions.

A considerably more advanced program timing tool that is part of the Mars[4] real-time operating system development environment (Pospischil et al 1992) places restrictions on allowable tasks that are reminiscent of Woodbury (1986). These are:

- Input at beginning.
- Output at end.
- No internal communication.
- No gotos.
- No recursions.
- All loops statically bounded.
- No dynamic memory allocation.

These limitations allow the timing tool to compute program time bounds from information supplied directly from the code. To make the timing bounds tighter, this timing tool employs programs that supply auxiliary information to permit more accurate calculations:

- Scope—code block with one entry, one exit.
- Marker—limit on executions of a scope.
- Loop sequence—interdependencies of loop bounds.

The Mars timing tool is embedded in a development environment that includes two editors, a compiler, and two off-line schedulers (Kopetz et al 1989; Pospischil et al 1992). These components are used in the iterative sequence:

- Text editor accepts programmer entry.
- Compiler generates timing tree structure.

---

[4] Mars (MAintainable Real-time System) is a research real-time system developed at Technische Universitat Wien (Vienna), specifically for investigating predictability of real-time systems.

- Timing editor allows speculative times for unknowns.
- MaxT[5] analysis inserts calculated times for knowns.
- Task and bus schedulers generate schedules.

Because of the graphical presentation of the editors (see Figure 1), the programmer is able to iteratively refine real-time task performance so that a feasible schedule (all tasks are scheduled and all deadlines are met) can be attained.

The Mars and the Park and Shaw timing tools are examples of automatically generated program timing information. It is also possible, but tedious, to prepare this information by manual analysis (Smith 1990).

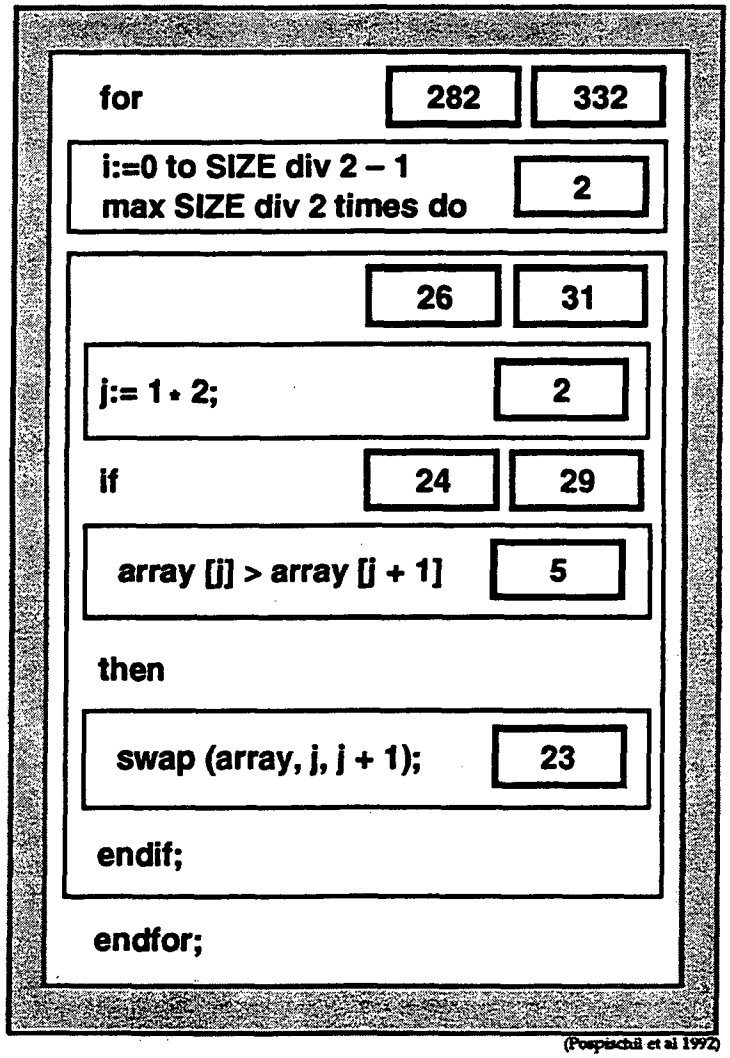


Figure 1. Mars Timing Editor Presentation.

[5] The maximum time a particular computation is expected to take.

In distributed systems, knowledge of local program time is insufficient to characterize overall behavior of the system and to schedule node-to-node communications relative to individual node activity. This is because clock skew between system nodes is unavoidable, and time can only be locally ordered[6] (Lamport 1978). This broader view of interacting modules was called "system timing analysis" earlier in this paper. Clock skew and partial time ordering cause confusion between nodes (if no corrective measures are taken) of the following types:

- Absolute times of event.
- Causal ordering of events.
- Calculation of time intervals.
- Data consistency (timestamps).
- Uncoordinated scheduling of tasks on separate processors.

Lamport suggested a method of ordering "events" on a global scale by exchange of timestamps in messages so that there was a relation to physical time within an error bound, even though individual computer crystal clocks drifted relative to each other. The method involves using timestamps in messages that are more or less regularly exchanged to make positive corrections to local clocks. Negative corrections are not allowed, since this would make local clocks non-monotonic in some instances. Later work by Kopetz and Ochsenreiter (1987) uses a network interface with an embedded timestamper (timestamps are automatically added to outgoing and incoming messages) to maintain clock synchronization between physically separated nodes on a network. An algorithm is proposed for calculating clock corrections that the authors maintain is good to about 100 μsec while utilizing about 1% of CPU time. Both of these methods will ensure that timestamps from anywhere in the system are within some small error of the correct universal (e.g., astronomical) time, but they may impose an order on concurrent events within the error window that is unrelated to causality (Fidge 1991). In some real-time systems this may not be a problem, because causality is assumed by other logic, or a consistent total ordering is all that is necessary for resource allocation algorithms or deterministic communication scheduling.

In systems where exact causal ordering is required, Fidge (1991) has proposed "logical clocks" by which causal ordering or concurrency[7] can be exactly detected. The Fidge method uses a "vector" clock representation of time by which communicating tasks retain a vector of last-known local clock times of all tasks with which they interact, either directly or indirectly. Updates occur at message interchanges. By comparing other tasks' ideas of time, a task can make decisions about event orderings in remote parts of the system. Logical clocks maintain an order consistent with causality, but not necessarily precisely in step with astronomical time. Some time intervals may contracted or extended.

A global theory of time in distributed systems is important for the previously mentioned scheduling purposes, and also for ensuring that synchronization occurs properly between cooperating portions of the system. This is generally the main concern in communication protocol design (Preckshot 1993), and some communication protocol techniques for validating specifications are used in real-time system analysis. In particular, reachability analysis has been combined with symbolic execution (to eliminate infeasible states) to explore such synchronization problems as deadlock, livelock, unspecified messages, and state transition difficulties (Young and Taylor 1988).

---

[6] Resulting in system-wide partial ordering.

[7] Two concurrently executing tasks detect events but do not exchange messages. It is impossible, in the absence of message interchange, to determine event ordering except that the events occurred in the same interval.

Summarizing real-time system timing issues: with suitable restrictions, real-time code timing can be predicted within deterministic bounds; computer hardware exhibits bounded indeterminacy; there are methods for logically and physically synchronizing distributed systems within a small time error.

3.3 Scheduling

The final kind of timing analysis, schedulability and priority analysis, has to do with *when* and *in what order* tasks are executed. Once code timing is known, or (with something like the Mars timing editor) code timing estimates are known, a tentative schedule can be constructed if static scheduling is being used. Static scheduling has the very distinct advantages that the method used to calculate the schedule is not a significant factor during runtime, and the schedule (or schedules) can be validated before use. Some dynamic scheduling algorithms may themselves use substantial CPU resources, and so contribute to the problem they attempt to solve.

If it were just a matter of concatenating code runtimes and checking that sufficient CPU cycles were available, scheduling would be easy. Unfortunately, if real-time code is at all complex, some code talks to other code or uses the same resources. At the very least in distributed systems, the use of interconnecting communications media must be arbitrated. In predictable systems, this means that it must be possible to predict that all necessary messages will be delivered in time to allow recipient code to meet deadlines or, more restrictively, to determine what messages will be sent and when they will be sent. The most definitive and direct course is to lay out task runtimes, resource usage intervals, and communication channel usage, and demonstrate that there are no conflicts. This is the approach taken in static scheduling, and is exemplified by the Mars scheduling approach (Kopetz et al 1989). The Mars system permits several previously calculated schedules to be used one at a time, depending upon system needs.

Since the scheduling algorithm used for statically scheduled systems is immaterial, provided it produces workable previously calculated schedules, algorithms reported in use in dynamically scheduled systems will be discussed without further mention of static systems. As previously mentioned, the rate monotonic algorithm is optimal among fixed-priority scheduling algorithms, and produces a least upper bound on processor utilization of:

$$U(n) = n(2^{1/n} - 1)$$ where n is the number of tasks (Liu and Layland 1973).

Other algorithms are in use besides the rate monotonic algorithm. Zhao et al (1987a) state that the general optimal scheduling problem (scheduling randomly arriving tasks with indeterminate computation times, deadline and resource constraints) is "NP-hard" (time to solve increases faster than any finite power of n, the number of tasks). Consequently, Zhao proposes a heuristic that produced a satisfactory but not optimal schedule in several examples given in the paper.[8] It is instructive to use this as an example of techniques used in scheduling. Zhao represents the task set Ω composed of tasks τ_j to be scheduled as characterized by:

1. Processing time C_j
2. Deadline D_j
3. Resource requirements R_j, where $R_j = \{R(p), R(q),...R(z)\}$
4. $R_j(i) \in R$, a set of r resources
5. $1 \le p \le q \le z \le r$ are indices in R associated with τ_j.

[8] It cannot be proved that a heuristic algorithm will always be correct, but in statically scheduled systems it is only necessary to prove that the resulting schedule is correct.

The set of non-preemptable tasks is scheduled on a series of time slices S_k which occur at task completions (or in later work, also at preemptions (Zhao et al 1987b)). Time slices are similar to the familiar time slice of time-sharing operating systems, except that they do not occur except at exceptions, completions, or preemptions. A schedule is full if for all j, the task τ_j processing time C_j is less than the sum of all time slices in which τ_j executes. A partial schedule includes tasks which have insufficient slice time to complete. A schedule is fully feasible if for all j, the schedule is full and the end of the last slice time S_k for each task τ_j is less than D_j. A fully feasible schedule is built by starting with a null schedule and extending it one task at a time. Each extension is partial until a full schedule is reached. The first task is selected using smallest-laxity-first (laxity is deadline minus current time minus computation time), after which the heuristic algorithm is used to select additional tasks. The authors define conditions called strong- and weak-feasibility, with a proof that weakly feasible[9] schedules cannot be extended to full feasibility, resulting in a search cut-off heuristic. They provide a formula for determining strong feasibility of partial schedules.

The Zhao technique illustrates two important points about dynamic scheduling algorithms. The first is that when task τ_j arrives, there may already be a schedule in place, so the algorithm must provide some method of extending an existing schedule. The second is the dilemma posed by optimal versus heuristic algorithms. An optimal algorithm may in theory be able to schedule a task set, but not in time to do any good. A heuristic algorithm may work quickly enough, but may fail to find a feasible schedule.

The rate monotonic algorithm has been extended to systems with periodic and aperiodic tasks with priority inversion (Sha and Goodenough 1990; Sha et al 1990; Warren 1991). In simple rate-monotonic scheduling, the priority of periodic task τ_j is inversely dependent upon its period T_j, and higher-priority tasks preempt lower-priority tasks. This method works for independent tasks, but independence is rarely the case in distributed, communicating real-time systems. Tasks interact by communication and resource conflicts. In preemptive systems, a condition known as priority inversion occurs when a lower-priority task blocks a higher-priority preemptive task (and thus deadlocks) by possessing a resource the higher-priority task needs when preemption is attempted. The situation is resolved by having the lower-priority task inherit the priority of the preempting task while it is in the critical section of code (Sha et al 1990). Schedulability calculations are extended by adding the appropriate blocking time to the schedulability relation as:

$$C_1/T_1 + C_2/T_2 \ldots + C_n/T_n +$$
$$\max(B_1/T_1, B_2/T_2, \ldots, B_{n-1}/T_{n-1}) \leq U(n) = n(2^{1/n} - 1)$$

where:

- n is the number of tasks
- C_i is the execution time for task τ_i
- B_i is the longest blocking time for task τ_i[10]
- T_i is the period time for task τ_i
- U(n) is the least-upper-bound CPU utilization for n tasks and priority is ordered highest = 1 to lowest = n.

---

[9] Weak feasibility is a mathematical condition of an incomplete schedule and tasks remaining to be scheduled that indicates that no further tasks may be added without violating deadline constraints. The details are beyond the scope of this report.

[10] From a system response standpoint, blocking times should be kept as short as possible, even though the schedulability relation is satisfied.

Sha and Goodenough present an additional mathematical test (not repeated here) for determining if a periodic task set will meet its deadlines for all possible task phasings. A problem remains that aperiodic tasks are not included in the theory. The theory is extended to high-priority aperiodic tasks of bounded frequency and execution time by scheduling an "aperiodic server" that is, in fact, a fictitious periodic task with period equal to the frequency bound of the aperiodic task. Provided that the computation time of the aperiodic task is well within the CPU allocation for the aperiodic server, random preemptions whose frequency does not exceed the frequency bound will be serviced rapidly.

The "Spring Kernel"[11] (Stankovic and Ramamritham 1991) is an approach that uses a combination of dynamic and static scheduling. Tasks are divided into critical, essential, and non-essential tasks, with critical task deadlines being guaranteed by previously calculated static schedules. Essential tasks are dynamically scheduled by an on-line scheduler that attempts to guarantee deadlines by using auxiliary information not usually available to general-purpose operating system schedulers. Stankovic and Ramamritham suggest that because real-time systems are more tightly constrained than general-purpose computing systems, more is known about tasks in the system and should be used to advantage by the scheduler. The kinds of auxiliary information they propose are:

- An equation for worst case execution time.
- Deadlines, periods, or other real-time constraints.
- Priority.
- Resources needed.
- Importance.
- Whether the task is "incremental."
- Precedence graph (of interacting tasks).
- Communication graph (of communicating tasks).

An incremental task is a special case of an imprecise calculation. An imprecise calculation is a less-accurate or heuristic calculation that is used to produce some sort of result when a more-precise but run-length unpredictable algorithm fails to produce timely results. Imprecise calculations are used to ensure that task calculation times are bounded. Incremental tasks produce imprecise results early in the process, but continue to refine the results until done, or a calculation deadline is exceeded. In the Spring Kernel approach, the combination of numerical expressions for task run time and imprecise calculation information give the scheduler a better chance to construct a feasible schedule.

The important points about scheduling are worth recapitulating:

- The scheduling algorithm used in statically scheduled systems is immaterial, provided it produces workable schedules.
- The most definitive and direct course is to lay out task runtimes, resource usage intervals, and communication channel usage, and demonstrate that there are no conflicts.
- $U(n) = n(2^{1/n} - 1)$ is a conservative least upper bound of CPU utilization for a set of n tasks.
- A dynamic scheduling algorithm must provide some method of extending an existing schedule.
- An optimal algorithm may, in theory, be able to schedule a task set, but not in time to do any good.

---

[11] The Spring Kernel is a research real-time operating system developed at the University of Massachusetts at Amherst to investigate predictability and performance attributes of large, complex real-time systems.

- A heuristic algorithm may work quickly enough, but may fail to find a feasible schedule.
- The rate monotonic algorithm has been extended to systems with periodic and aperiodic tasks with priority inversion.
- More is known about tasks in real-time systems than in general time-sharing systems, and can be used to advantage by dynamic schedulers.
- Imprecise calculations can be used to ensure that task calculation times are bounded.

3.4 Prototyping

The issue of what prototypes to build and when to build them is complicated by the variety of purposes that prototypes serve. For convenience, an expanded form of earlier discussion of prototype kinds (Taylor and Standish 1982) is presented here in list form:

1. Feasibility prototype (do not know how to build it).
   1a. Abstract requirements but too difficult in practice.
   1b. Correct requirements but do not understand environment.
2. Requirements prototype (do not know what to build).
3. First of type off line.

The danger of all prototypes is that, driven by economic pressures, the prototype becomes the product. This is particularly true of software prototypes—they tend to be thrown together in a hurry, like temporary military buildings, and then amaze everyone by a rickety survival ten times longer than anyone expected. There is therefore some hazard in requiring prototypes without carefully prescribed purpose or limits to usage.

Kind 2 software prototypes are usually built either to get user feedback on how a prospective system should function, or to try several known approaches to a problem to see which suggests the most cost-effective solution. The most likely usage in safety-critical embedded systems (such as reactor protection systems) is for human-factors investigation of surveillance difficulty and maintainability. Safety-related annunciation systems can also be reviewed in prototype form to ensure that operators are not confused by reports from the display system during stressful situations. In the former case, an actual physical prototype with minimal software would be needed, since surveillance and maintenance access requires manual interaction. In the latter case, artwork mockups and hand-replacement of imitation displays might be sufficient. Since there is substantially more user interaction with plant control and process parameter display systems, Kind 2 prototypes for these systems might be considerably more comprehensive. For reactor protection and reactor control systems, it is probably advantageous to consult human factors specialists to determine the extent of Kind 2 prototypes required.

Kind 1a prototypes are built because, while system requirements may be exact, they are sometimes too abstract to engender a practical response. An example would be, "design an automobile-mounted system that avoids automotive collisions." Everyone understands exactly what is wanted, but nobody can build it because they do not know how. The situation pertains in part to reactor protection systems, because the reliability requirements for software may be unattainable in general, or a new reactor design may introduce new physical behavior that protection system software must track. In the first case, constructing a software prototype to achieve modest gains in reliability may be useful, but where requirements are significantly beyond the limits of practice, adjusting requirements to more realistic levels may be the only reasonable course. In the second case (a core protection calculator for a new kind

of core, for example), the requirement is known (trip the reactor if the core approaches safety limits), but the prototype is necessary to work out the protection algorithm details.

Kind 1b prototypes probably will be useful in all real-time computer systems associated with nuclear reactors. A case has been described (bounded indeterminacy in computer systems (Park and Shaw 1991)) in which measurement of the hardware environment provides significant information for predicting performance. Additional areas include communication system indeterminacy, sensor readout random variations, and computer interface characteristics. Kind 1b prototypes usually involve hardware and test software for collecting data needed early in the design and performance prediction process. For this reason, they are more useful early in the design process rather than later, when the news may be too late for economic correction.

The final kind of prototype in the above list, Kind 3, is typically used for validating that the production process has produced something that meets the original specifications. For computer-based protection systems and control systems, these prototypes would generally be sufficient portions of a final system for software–hardware integration and validation testing. For ultra-reliable systems, there is some sentiment that the validation prototype must be the final system, since no change in software can be tolerated without fatally disturbing the validation process. This may not be the case for performance issues, however, so this paper leaves ultra-reliability issues to Lawrence (1992) and Barter and Zucconi (1992). The performance objectives of final prototypes are to show that:

- The performance models used match the product.
- Performance scales as predicted.
- Unforeseen performance "glitches" are unlikely.
- Performance equals or exceeds requirements.

Because models play such a significant role in real-time system development, decisions about final prototypes necessarily should be guided by model development and predictions. The model is often a useful guide to how much of the final system must actually be prototyped to resolve outstanding performance questions. These issues will be covered in more detail in Section 3.6, Modeling and Measurement.

3.5 Development Approaches

Development of real-time systems was originally an ad hoc process and, unfortunately, it still is for some developers. Tools used for development include the "In-Circuit Emulator" (ICE),[12] that allows the software engineer to follow code execution at the machine language level. There are software tools to support ICEs that allow the engineer to follow code execution at source language level, but there is still a regrettable tendency to resort to assembly language much too often. This paper will not further consider such approaches since the risks involved exceed what is tolerable for reliable reactor protection systems. The use of ICEs is not ruled out, but should be limited to specific performance problems where especially tight timing must be demonstrated, or where performance is being verified. In other words, high-level language based development method, timing analyses, and proof of schedule correctness are preferred over code development at the machine level.

---

12 The ICE is a very commonly used electronic tool available from computer system vendors and independent equipment suppliers.

3.5.1 Tuning Performance During Integration

A systematic approach that was popular in the 1980s, and still is in some quarters, can be described as "make it right, then make it go fast." Developers use some software development method that helps to produce reliable code, and then discover code execution times by experiment. A software tool called a "profiler" is used to determine which modules are getting most use ("hot spots") and these modules are rewritten if necessary to speed up system execution. ICEs may be used in place of profilers in embedded systems that have limited accessibility.

This method works for systems of limited extent, but can run into trouble if performance problems are dominated by poor system architecture rather than isolated code module performance. It also has the defect that performance problems are not discovered until late in development, when little time (and money) remains to correct them.

3.5.2 Engineering Performance During Design

More recent methods include specifying performance requirements early in the design so that appropriate system architectures can be chosen and performance requirements can be allocated between parts of the architecture. This is not an easy thing to do *a priori*, so most such methods include iterative modeling techniques to work out design alternatives.

An example of this approach is provided by Smith (1990), who calls her methodology Software Performance Engineering (SPE). SPE has not been used for hard real-time systems,[13] but a description illustrates the iterative modeling and measurement sequence well. Initially, requirements are captured without constraints. Requirements analysis confirms whether or not the requirements are feasible, and suitable designs are identified. In a pattern that will be repeated throughout the design cycle, best- and worst-case scenarios are developed. Early performance models are devised and exercised to predict best- and worst-case performance, and the design is modified as necessary to bring worst-case performance within requirements. As design progresses through implementation, more and more of the model(s) is (are) replaced by data obtained from actual performance measurements, and both the model(s) and the design are iteratively corrected to match actual performance and to meet performance goals.

The methodology described by Kopetz (Kopetz et al 1991) used in the Mars development environment follows a similar scheme, except that deterministic rather than statistical timing constraints are used. The Kopetz method models real-time systems as a set of real-time transactions (a stimulus–computation–response event string) that is subject to deadlines. Transactions are broken down to tasks and distributed to physical nodes, which results in a set of performance requirements for both tasks and communications between tasks. Using the previously mentioned text and timing editors (Pospischil et al 1992) (Section 3.2, Timing) and static scheduling software (Section 3.3, Scheduling), an iterative process of refinement is followed until a set of predictable tasks and a feasible static schedule have been implemented.

In both approaches just mentioned, the methods reach the validation stage with well-developed performance models, which are aids to choosing the appropriate late-stage prototype and devising performance tests. From a performance validation standpoint, performance models are quite helpful because they afford more insight into the system being scrutinized.

---

[13] The usual application domain is transaction-type soft real-time systems where statistical throughput measures are acceptable.

3.5.3 Early Prototyping

Another school of thought proposes early or "operational" prototyping (Davis 1992). This is actually a staged or "versioned" stepwise development approach in which progressively more capable systems are developed and released as "versions" or "major releases." Since early versions have little capability, it is usually easy to achieve performance goals. The stepwise approach distributes the effort and risk of meeting performance objectives over a longer time, during which more can be learned about the problem, the hardware, and the environment. The method is favored in situations where there are substantial unknowns, insufficient money for full system development, and vague requirements. It is the de facto method for much commercial software, real-time or not. Early versions have the reputation of being quite "buggy." Staged implementation may not be a good choice for reactor protection systems because substantial unknowns and vague requirements are not tolerable in safety-critical systems. Commercial software products that use this development approach should be used with extreme caution in safety applications.

3.5.4 Special Languages

Specially designed computer languages have been proposed to make the development of real-time systems easier or more accurate. Some of the more extravagant claims suggest that correct language design will make it difficult to write incorrect code. To date, that has been the ideal rather than practice. Real-time computer languages can be split roughly into two subgroups: specification languages and implementation languages. Some examples of the former are Task Sequencing Language (TSL) (Rosenblum 1991), Extended Temporal Description Language (TPDL*) (Cabodi et al 1991), RT-ASLAN (Auernheimer and Kemmerer 1986), and various Formal Description Techniques (FDTs) used in communication protocol specification (Preckshot 1993). Examples of the latter are Occam 2 (Burns and Wellings 1990), Modula II (Burns and Wellings 1990), Flex (Kenny and Lin 1991), and the U.S. Department of Defense Ada (Leveson et al 1991; Burns and Wellings 1990; Clapp et al 1986).

Specification languages attempt to provide an unambiguous way to describe timing and sequencing specifications. Some of them have been translated into models that can be utilized to perform reachability analysis (Preckshot 1993) and thus eliminate certain synchronization hazards. It is still necessary, once a system has been described in a specification language, to convert the specification language form to an executable language form. The practical effect of these specification languages on performance is unclear, although having an unambiguous timing requirement must be counted a plus.

Real-time implementation languages usually try to provide parallelism and synchronization constructs. Burns and Wellings (1990) list three general areas:

- Concurrent execution through the notion of processes.
- Process synchronization.
- Inter-process communication.

Other constructions include monitors (Hoare 1974), mutual exclusion markers (Holt 1983), explicit parallelism, and rendezvous mechanisms such as that used in Ada. Occam 2 is unusual in that it assumes language statements can be executed in parallel unless specifically told they must be executed in sequence. The implementation languages are more concerned with getting concurrent program interactions correct than with optimizing performance. In this respect they support the performance tuning approach at integration, rather than building performance in at design time. There is often the implicit assumption that more performance is available through greater parallelism, but explicit support for deadlines is not generally the case. Ada and Flex support exception mechanisms that allow imprecise calculations to be substituted for tasks that are close to exceeding a deadline. Flex has some

characteristics of a specification language because code blocks are enclosed in "constraints," which act like timing specifications. The exception approach still requires assistance from an operating system or runtime kernel.

3.6 Modeling and Measurement

Modeling is a necessary discipline when building real-time systems. Models serve as performance predictors during initial development stages, and later as guides for validating the system. One of the most significant models in use, the static schedule, may not be recognized as a model because, like the "purloined letter," it is too obvious. Static schedules are exact models for how a statically scheduled real-time system is supposed to perform, and can be used directly to verify that software execution and communication occur precisely as planned.

The other models generally in use for real-time system development are code execution models, system-level Petri net models, and queuing network models. Code execution models were previously discussed in Section 3.2, and they produce code timing data that is used by the system-level models. This section will discuss variants of the other two system model types and methods of measuring software performance for comparison to model data.

Petri nets (Peterson 1977) are a method of representing systems that go through complex but finite state transitions. They are usually represented graphically as a directed graph of three types of objects (places, transitions, and directed arcs) and a "marking" that describes the system state. Figure 2 shows such a net, with the marking being the solid dots, which are called "tokens." Mathematically, the simple Petri net is stated:

$$PN = (P, T, I(t), O(t), M)$$

where

P is a set of places,
T is a set of transitions,
$I(t_j)$ is the set of input places for transition t_j,
$O(t_j)$ is the set of output places for transition t_j,
M is the marking.

The model is executed by firing transitions, which results in a new marking. The sequence of markings obtained by transition firings represents a trajectory through the state space of the system being modeled. In the simple Petri net, transitions may fire as soon as they are enabled (at least one token is in each input place of the firing transition). The new marking is determined by removing one token from all input places of the fired transition and placing a token in all output places of the fired transition. The reachability set of a Petri net is the set of all markings that can be reached by execution from an initial marking M_0. Note that because there may be ambiguities about which transition will fire when two transitions share the same input place (a "conflict"), determining the reachability set may be non-trivial or indeterminate. Finding the reachability set is called reachability analysis and is equivalent to finding the states that a correctly modeled system can assume. Ambiguities, non-conservation of tokens, more than one token in an input place, dead-end markings, and unexpected markings are all findings that may indicate logical problems with the original system but may also be modeling errors.

The original definition of the Petri net included no concept of time. Petri nets have been extended to timed Petri nets, Stochastic Petri Nets (SPNs), and Generalized Stochastic Petri Nets (GSPNs) (Conte

and Caselli 1991). GSPNs illustrate the method of extension quite well. The mathematical description of the GSPN is:

GSPN = (P, T, $\Pi(t)$,I(t), O(t), H(t), W(t), M),

where

P is a set of places,
T is a set of transitions,
$\Pi(t_j)$ is the priority of transition t_j,
$I(t_j)$ is the set of input places for transition t_j,
$O(t_j)$ is the set of output places for transition t_j,
$H(t_j)$ is the set of inhibitory places for transition t_j,
$W(t_j)$ is the rate or weight of transition t_j,
M is the marking.

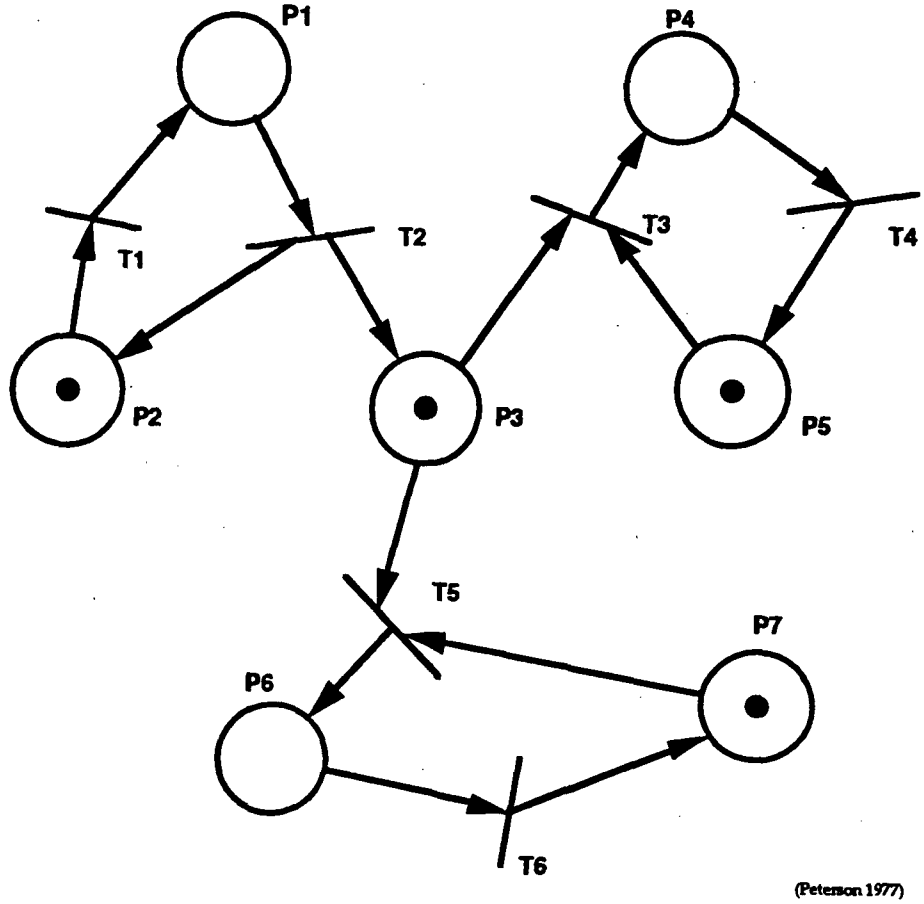


Figure 2. A Petri net Showing Conflict Between Transitions T3 and T5.

Two new features for arbitration of conflicts, Π(t) the transition priority, and W(t) the rate or weight, are added. W(t) can be a probability distribution function for firing delay, a simple firing delay, or a probabilistic weight to be used to select which of several enabled but conflicting transitions will fire. H(t) now allows modeling of inhibitory behavior (a token prevents firing) as well as permissive behavior. The augmented Petri net models can model timed behavior, but still do not directly model queuing behavior, which is often explicitly coded into real-time system designs. Petri nets may also be executed directly (as a simulation) or reachability graphs can be converted to a continuous-time Markov chain model, from which statistical performance indices are then computed.

Queuing network models (Lazowska et al 1984) were among the first models to be used to represent computer systems that provided services to a user program population that was known only statistically. Figure 3 shows a simple queuing model of a single CPU and four disks that can be used in parallel (MacDougall 1987). Associated with each queue is a service time (usually modeled probabilistically) and, where multiple output paths are possible, a probability for selection of each path. Work coming into the system is described as a stochastic generation process; system throughput

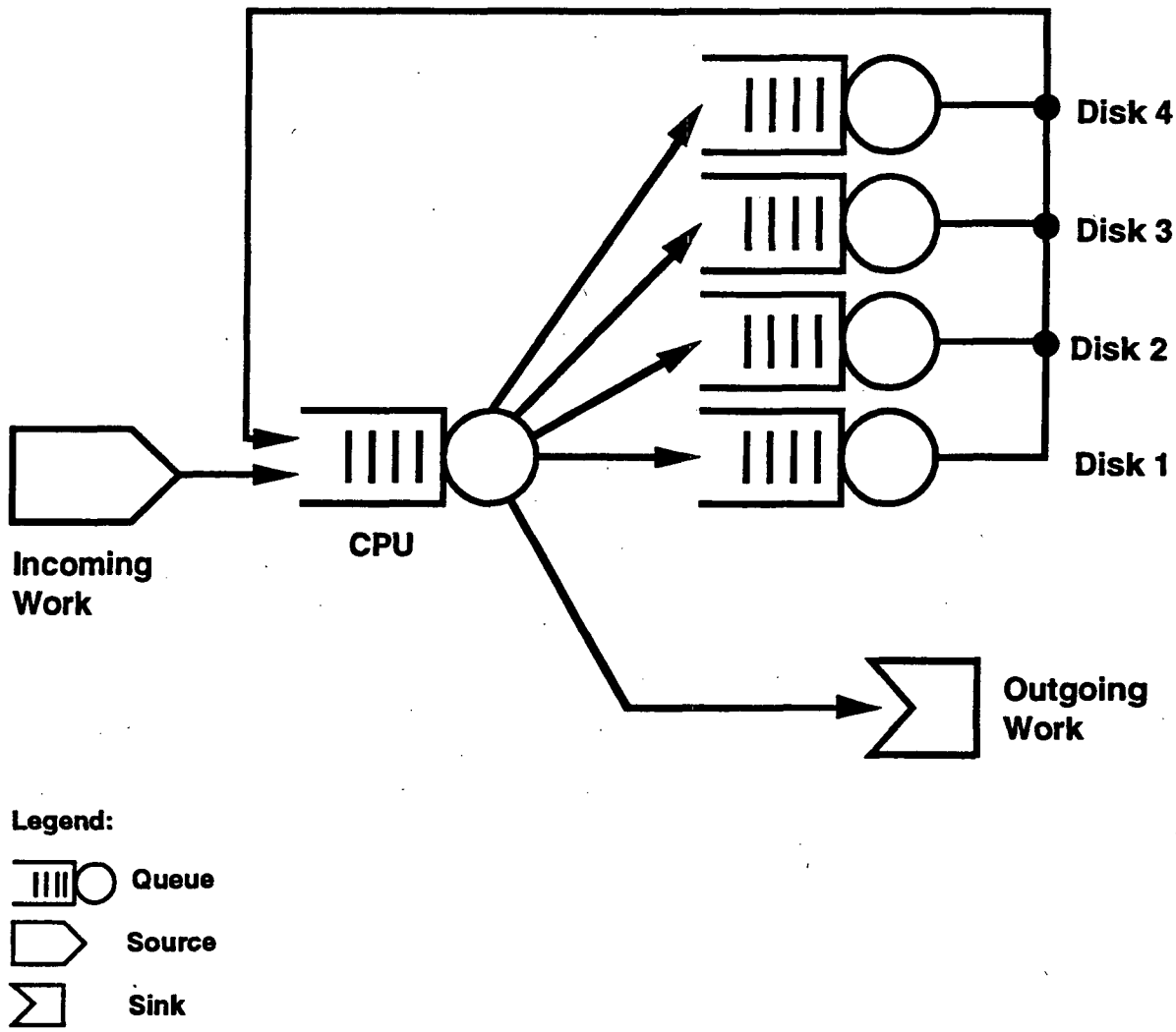


Figure 3. A Simplified Queuing Network Model of a Computer System with Four Disks.

response, including queue loadings, are some model outputs. Like Petri nets, queuing network models can be used for simulation or solved analytically. They are often used (Smith 1990) for performance analysis of systems for which average or expected throughput under specified load conditions is desired. There are two defects when used for analysis of hard real-time systems: average performance is not acceptable when there are hard deadlines, and model output is very dependent upon accurate workload characterization. In spite of this, these models may still be useful to understand system operation if deadlines are guaranteed by other logic. In particular, performance predictions may serve both as initial feasibility confirmation and as later testing yardsticks.

An obvious question is, "Why not combine Petri nets and queuing network models?" The Timed PQ Net (Chang et al 1989) is such a combined model. Queues have been added to transitions, which now obscure the location of marking tokens.[14] No significant usage has been reported with this model yet.

Once code has begun to be implemented, continuous comparison with model predictions is necessary; otherwise there very likely will be a steadily widening gulf between planned performance and actual performance. Measurement techniques are needed during development and during integration to demonstrate performance to developers and regulators. From Smith (1990), measurement is usually performed in four situations:

1. Measurement data from existing systems.
2. Measurements of experiments or prototypes.
3. Measurement of actual code units on different hardware and then extrapolating to the target system.
4. Measurement of the target system.

The measurement situation often governs which measurement tools are possible or available. The tools generally available are:

- System monitors (e.g., task resource and time usage).[15]
- Program monitors (profilers).
- System event recorders.
- External measurement devices (logic analyzers, ICEs).
- Instrumented programs.
- Benchmarks.

System monitors and program monitors are tools that need support from some sort of operating system. The difference between the two is that program monitors provide more detailed information about the internals of program operation, and often need access to source code and a syntactical structure chart. System event recorders can be software or hardware devices that note the time of predefined "events." Loggers would fit this definition, and are often required on real-time control systems to maintain a record of "significant" actions. All three of these measurement methods are better suited to established systems or systems with substantial operating system support (situations 1, 3, and 4 above).

External measurement devices have the advantage that they generally do not interfere with running code, but merely observe it. This is balanced by the difficulty of application, since invasive hardware procedures are required, and it is hard to tie the machine language instruction stream to source code without coupling with the compiler and linker. In spite of these difficulties, external measurement is often used in situations 2 and 4 above because no acceptable alternatives exist.

---

[14] Tokens may now exist in a queue in a transition or in a place, which violates the initial precept of Petri that places contain the tokens.

[15] These are not monitors as proposed by C.A.R. Hoare (1974).

Instrumented programs, which are programs with extra performance measurement code compiled into them, are used in all situations and in combination with other measurement techniques. The simplest form of instrumentation is loop counting or clock watching, and it is very effective. Probably the highest frequency of use for instrumented programs is in unit performance testing. Execution time of a code module can be easily measured by reading the system clock, executing the module N times, and rereading the system clock. Module time is then the elapsed time divided by N. A special form of instrumented program is called a "benchmark," which is a calculation performed N times to measure processor speed for particular kinds of computations. Much has been written about benchmarks, mostly obscuring their real effects because of advertising hyperbole. Regardless, benchmarks are useful point measurements when specific types of calculations are important to real-time performance.

3.7 Real-Time Operating Systems

Real-time computer systems are usually designed in one of two ways. The first way is to design the entire system from scratch, writing all of the code that ends up in the target computer system as a dedicated application or coordinated group of applications. The second way is to separate the applications from commonly used "system services" and either to write or acquire a "real-time operating system" to provide these system services. Real-time operating systems are also called "embedded kernels" or "real-time kernels," and as the names suggest, are the core around which the applications are built. It is very likely that real-time operating systems will be encountered sooner or later in nuclear power plant computer systems. This section will discuss the general features of some commercially offered real-time operating systems, as well as two research systems reported in the literature.

Because of the number and variety of offerings calling themselves "real-time operating systems," it is beyond the scope of this work to review even a portion of them. To name several, there are VxWorks and the Wind River Kernel (Wind River Systems, Inc.), OS-9 and OS-9000 (Microware Systems Corporation), VMS and VAX ELN (Digital Equipment Corporation), VRTX (Ready Systems), LynxOS (Lynx Real-Time Systems, Inc.), and UniFLEX (RTMX-UniFLEX). These systems share some generic features, among which the most consistent seems to be "configurability." In order to satisfy the largest possible number of potential customers, real-time operating system vendors often supply a system that can include or exclude a significant fraction of the advertised "features." This reconfigurability can make performance modeling more difficult because of uncertainty about how the reconfiguration is accomplished and what the effect of each feature is upon performance. The other features generally supplied are multi-tasking, scheduling, exception handling, device drivers, mathematical and other libraries, compilation, linking and loading services, communication services, debugging tools, performance measurement tools, and a software development environment. These features are so numerous and varied that their effects on real-time performance and software reliability can only be reviewed on a case-by-case basis.

Two operating systems that have been reported in the literature are significant because they support a requirement that the resulting system shall be predictable. These two are Mars (Kopetz et al 1989) and the Spring Kernel (Stankovic and Ramamritham 1991). Mars is a statically scheduled system that has support for mechanized timing analysis and schedule generation. Mars is a distributed system that is linked together by Time Domain Multiplexed (TDM) data links (Preckshot 1993). The communication system is state-based (see Preckshot 1993) and state messages are unconsumed when read. Timestamps associate a "validity time" with each state message, and fault tolerance is assisted

by not acting upon stale state data. A global time base is maintained within a small error window that permits absolute timing of events, causal ordering, accurate calculation of time intervals, and consistency between a real-time database and the environment. The Mars system is designed to eliminate uncertainty at the expense of some flexibility.

The Spring Kernel is designed to allow somewhat more flexibility than the Mars system, but at the expense of predictability of some parts of the system. Real-time tasks are designated as "critical," "essential," and "non-essential." Critical tasks have bounded execution times and are statically scheduled. Essential and non-essential tasks are dynamically scheduled, with essential tasks forming a schedulable subset. The dynamic scheduler uses auxiliary information about tasks to enhance scheduling success. Imprecise calculations are used to limit essential task execution times where necessary. Non-essential tasks are delayed or aborted if insufficient time is available to complete all tasks to be scheduled.

The Mars system and the Spring Kernel illustrate two approaches to predictability using a real-time operating system. Commercially available real-time operating systems may have equivalently convincing methods of ensuring predictability, but demonstration of this depends upon the operating system vendor.

References

Auernheimer, Brent, Kemmerer, Richard A., September 1986, "RT-ASLAN: A Specification Language for Real-Time Systems," *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 9, pp. 879–889.

Barter, R., and Zucconi, L., 1992, "Verification and Validation Techniques for Software Safety and Reliability," Lawrence Livermore National Laboratory, to be released.

Burns, Alan, and Wellings, Andy, 1990, *Real-Time Systems and Their Programming Languages*, Addison-Wesley Publishing Company.

Cabodi, G., Camurati, P., Prineto, P., Sonza Reorda, M., April 1991, "TPDL*: Extended Temporal Profile Description Language," *Software Practice and Experience*, Vol. 21(4), pp. 355–374.

Callison, H. Rebecca, Shaw, Alan C., April 1991, "Building a Real-Time Kernel: First Steps in Validating a Pure Process/Adt Model," *Software Practice and Experience*, Vol. 21(4), pp. 337–354.

Chang, Carl K., and four co-authors, March 1989, "Modeling a Real-Time Multitasking System in a Timed PQ Net," *IEEE Software*, pp. 46–51.

Clapp, Russel M., Duchesneau, Louis, Volz, Richard A., Mudge, Trevor N., Schultze, Timothy, August 1986, "Toward Real-Time Performance Benchmarks for ADA," *Comm. ACM 29*, No. 8, pp. 760–778.

Conte, Gianni, and Caselli, Stefano, 1991, Generalized Stochastic Petri Nets and Their Use in Modeling Distributed Architectures, *IEEE CH3001-5/91/0000/0296*, pp. 296–303.

Davis, Alan M., September 1992, "Operational Prototyping: A New Development Approach," *IEEE Software*, pp. 70–78.

Fidge, Colin, August 1991, "Logical Time in Distributed Computing Systems," *Computer*, pp. 28–33.

Gopinath, Prabha, Bihari, Thomas, Gupta, Rajiv, September 1992, "Compiler Support for Object-Oriented Real-Time Software," *IEEE Software*, pp. 45–50.

Haase, Volkmar H., September 1981, "Real-Time Behavior of Programs," *IEEE Trans. on Software Engineering*, Vol. SE-7, No. 5, pp. 494–501.

Hoare, C.A.R., October 1974, "Monitors: An Operating System Structuring Concept," *Comm. ACM 17*, 10, pp. 549–557.

Holt, R.C., 1983, *Concurrent Euclid, the Unix System, and Tunis*, Addison-Wesley Publishing Company.

Kenny, Kevin B., Lin, Kwei-Jay, May 1991, "Building Flexible Real-Time Systems Using the Flex Language," *Computer*, pp. 70–78.

Kopetz, H., and six co-authors, February 1989, "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach," *IEEE Micro*, pp. 25–40.

Kopetz, H., circa 1990, "Event-Triggered Versus Time-Triggered Real-Time Systems," Technische Universitat Wien, Institut Fur Technische Informatik.

Kopetz, H., Ochsenreiter, W., 1987, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Trans. on Computers*, Vol. C-36, No. 8, pp. 933–940.

Kopetz, H., Zainlinger, R., Fohler, G., Kantz, H., Puschner, P., Shutz, W., May 1991, "The Design of Real-Time Systems: From Specification to Implementation and Verification," *Software Engineering Journal*, pp. 72–82.

Lala, Jaynarayan H., Harper, Richard E., Alger, Linda S., May 1991, "A Design Approach for Ultrareliable Real-Time Systems," *Computer*, pp. 12–22.

Lamport, Leslie, July 1978, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM 21*, No. 7, pp. 558–565.

Lawrence, Dennis J., 1992, Subject: Software reliability guidance, Lawrence Livermore National Laboratory, to be released.

Lazowska, E.D., Zahorjan, J., Graham, G.S., and Sevcik, K.C., 1984, *Quantitative System Performance: Computer System Analysis Using Queuing Network Models*, Prentice-Hall.

Leveson, Nancy G., Cha, Steven S., Shimeall, Timothy J., July 1991, "Safety Verification of Ada Programs Using Software Fault Trees," *IEEE Software*, pp 48–59.

Liu, C.L., Layland, James W., January 1973, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal ACM*, V. 20, No. 1, pp. 46–61.

MacDougall, M.H., 1987, *Simulating Computer Systems: Techniques and Tools*, MIT Press.

Natarajan, Swaminathan, Zhao, Wei, September 1992, "Issues in Building Dynamic Real-Time Systems," *IEEE Software*, pp. 16–21.

Oldehoeft, R.R., January 1983, "Program Graphs and Execution Behavior," *IEEE Trans. on Software Engineering*, Vol. SE-9, No. 1, pp. 103–108.

Park, Chang Yun, Shaw, Alan C., May 1991, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," *Computer*, pp. 48–57.

Peterson, James L., September 1977, "Petri Nets," *ACM Computing Surveys*, Vol. 9, No. 3, pp. 223–252.

Pospischil, Gustav, Puschner, Peter, Vrchoticky, Alexander, Zainlinger, Ralph, September 1992, "Developing Real-Time Tasks with Predictable Timing," *IEEE Software*, pp. 35–44.

Preckshot, George G., 1993, "Data Communications Guidance," Lawrence Livermore National Laboratory, to be released.

Rosenblum, David S., May 1991, "Specifying Concurrent Systems with TSL," *IEEE Software*, pp. 52–61.

Sha, Lui, and Goodenough, John B., April 1990, "Real-Time Scheduling Theory and Ada," *Computer*, pp. 53–62.

Sha, Lui, Rajkumar, Ragunathan, and Lehoczky, John P., September 1990, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. on Computers*, Vol. 39, 9, pp. 1175–1185.

Shaw, Alan C., July 1989, "Reasoning About Time in Higher-Level Language Software," *IEEE Trans. on Software Engineering*, Vol. 15, No. 7, pp. 875–889.

Smith, Connie U., 1990, *Performance Engineering of Software Systems*, Addison-Wesley Publishing Co.

Stankovic, John A., October 1988, "Misconceptions About Real-Time Computing," *Computer*, pp. 10–19.

Stankovic, John A., Ramamritham, Krithi, May 1991, "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, pp. 62–72.

Taylor, Tamara, Standish, Thomas A., December 1982, "Initial Thoughts on Rapid Prototyping Techniques," *ACM SIGSOFT Software Engineering Notes*, Vol. 7, No. 5, pp. 160–166.

Warren, Carl, June 1991, "Rate Monotonic Scheduling," *IEEE Micro*, pp. 34 –38, 102.

Woodbury, Michael H., 1986, "Analysis of the Execution Time of Real-Time Tasks," *1986 Proc. IEEE Real-Time Systems Symposium*, pp. 89–96.

Young, Michal, Taylor, Richard N., October 1988, "Combining Static Concurrency Analysis with Symbolic Execution," *IEEE Trans. on Software Engineering*, Vol. 14, No. 10, pp. 1499–1511.

Zhao, Wei, Ramamritham, Krithivasan, and Stankovic, John A., May 1987a, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Trans. on Software Engineering*, Vol. SE-13, 5, pp. 564 –577.

Zhao, Wei, Ramamritham, Krithivasan Stankovic, John A., August 1987b, "Preemptive Scheduling Under Time and Resource Constraints," *IEEE Trans. on Computers*, Vol. C-36, No. 8, pp. 949–960.

Zucconi, L., and Thomas, B., 1992, "Testing Existing Software for Safety-Related Applications," Lawrence Livermore National Laboratory, to be released.

Appendix B:
Real-Time Systems Complexity and Scalability

G.G. Preckshot

Manuscript Date: May 28, 1993

Contents

Abbreviations

COCOMO	COnstructive COst MOdel
CPU	Central Processing Unit
CSMA	Carrier Sensitive Multiple-Access
IEEE	Institute of Electrical and Electronics Engineers
MoD	Ministry of Defence (British)
NRC	Nuclear Regulatory Commission
PID	Proportional-Integral-Derivative
PLC	Programmable Logic Controllers
RADC	Rome Air Development Center
RISC	Reduced Instruction Set Computers
SCSI	Small Computer System Interface
SLOC	Source Lines of Code

Executive Summary

This report is the culmination of a study to identify system and software complexity metrics that will assist the Nuclear Regulatory Commission (NRC) in evaluating a system design. Complexity is the degree to which a system or component has a design or implementation that is difficult to understand and verify. To be effective, complexity measures must evaluate the system and the software implemented within the system. Using current tools and methodologies, determining system and software complexity before the system is built is very difficult, if not impossible. There are many tools and techniques available to measure complexity *after* a system is built, but making changes to the hardware and software at this point is difficult and expensive. Measuring complexity *during* the building process is the best way to determine and reduce unnecessary complexity.

Most of the quantitative measures of complexity have been designed to measure software, and therefore constitute the majority of the complexity metrics identified in this report. Currently, research is still directed at software complexity and software quality measures. Evaluation of system complexity relies mainly on qualitative evaluations. Detection and correction of complexity problems can be achieved if the products of each phase of development are evaluated before going on to the next phase of development. This report presents an approach for characterizing system complexity based on the attributes of the architecture and software components.

This report focuses on complexity and scalability. However, there are additional factors that affect the ability of the system to meet safety and performance requirements. Even with a good set of system and software requirements, many errors can be injected into the software due to the lack of good software engineering practices, and may remain because of insufficient testing.

There is no simple solution to evaluating software and system complexity. A starting point is the use of good systems engineering and software engineering practices in the design and implementation of a system. For software, the use of complexity metrics to identify and eliminate high-complexity components is a first step to reduce risk. The most popular software metric is the McCabe complexity metric, which indicates the complexity of low-level software modules. Industry generally agrees on a value for this metric that indicates acceptable levels of complexity and development risk.

Continual evaluation of performance and complexity of systems and software during each phase of development will result in the early identification of problems and cost-effective solutions.

Based on available literature and experience, the report concludes that there are no widely used metrics for software or system complexity. Problems also exist with adapting many complexity metrics to practical systems. McCabe and Butler note that practical systems normally have 100 times the design complexity of those used in textbooks.

Achieving NRC's goal will require a research effort to define metrics and to determine values that will indicate an acceptable hazard level, although little evidence suggests a reliable link between hazard and complexity. A combination of metrics and development standards may give NRC a firmer foundation on which to evaluate and approve systems.

Real-Time Systems Complexity and Scalability

1. Introduction

Real-time computer systems are essential parts of many modern control or protection systems, yet there is no generally accepted way to predict system complexity and performance. Questions like, "Will it work if we make it bigger?" "Is it safe?" "Is it fast enough?" "Does it work correctly?" have no ready answer. It is generally conceded that as such systems get "bigger," they become less safe, go slower, and make more errors. Even the notion of "bigger" has been hard to define because real-time systems are composed both of hardware and software and, while hardware components are easy to count, software has proved much harder to measure. The interactions between hardware, software, and the real world are even more difficult to quantify.

At the root of our conceptual problems is system complexity, for which there is no facile description. In addition, software adds a level of complexity to the traditional hardware-only system. Complexity has been approached by attempting to measure its aspects and their effects upon system qualities of interest. Most research effort has been directed to measuring one component of real-time systems, the software, because this component has historically been the cause of most uncertainty in system cost, reliability, and performance. The current state of that research is covered in the body of the report.

While attempts to measure software have proceeded slowly, industry and the software engineering profession have had to design and build complex real-time systems without benefit of research results. Two decades of software practice and experience have produced a number of indicators that tell the practitioner that a design is more or less complex. From experience, most designers know system attributes that make development and verification more difficult and make systems less reliable. Some of these attributes are presented later with justification from the literature. If no precise complexity measures are available, complexity attributes may still give an idea, albeit less precise, of the level and types of complexity present in a real-time system. To this end, an unvalidated taxonomy of real-time systems is introduced, which permits a rough classification of such systems into seven complexity levels.

Rightly or wrongly, small systems are considered simple and big systems are considered complex. Closely related to the idea of complexity is the idea of scaling, changing the size of one or more aspects of a real-time system. Scaling a real-time system can have unexpected effects on system response, reliability, and correctness. Some of the areas which historically have caused problems are covered in the body of the report. Scalability issues are important when a particular real-time system is not precisely adapted to a proposed application, as would happen when applying a commercial generic real-time control system to an actual industrial plant.

1.1 Purpose

The purpose of this report is to support the NRC's primary goal to estimate the hazard level of a system design. If the hazard level is found to be unacceptably high, the NRC may proscribe certain types of complexity for safety systems. It is desirable to identify quantitative measures or define a qualitative marker list of computer hardware architecture complexity to support this goal.

1.2 Scope

The report provides a summary of approaches and measures used by industry to evaluate system and software complexity. It includes evaluation methodology available to measure complexity for both systems and software, and presents approaches to defining system complexity and classifying systems according to complexity attributes. A discussion of system scalability identifies risk factors in developing large systems.

2. Complexity

Complexity is an indicator of how difficult a system, program, or module is to understand and maintain (Arthur 1985). The more complex a system, program, or module, the harder it is to analyze, test, and maintain.

System complexity and software complexity must be considered when judging the validity of a system design. System and software complexity measures are most often used to determine how difficult a design or design change will be to make and how long it will take to complete. They can also be used as indicators of how well a system has been designed. Systems, programs, and modules with high complexity levels are more likely to be delivered with errors than less complex ones that are easier to understand, maintain, and test.

When developing any system, an early prediction of system complexity is important. Estimating system risk early in the life cycle gives adequate time to redesign overly complex aspects of the system. When reviewing a system concept to determine if building the system as defined is reasonable, predicting system complexity and reallocating functions that have been deemed too complex saves time and money and produces a system with fewer complex components.

Ideally, system and software complexity metrics should be used throughout the development life cycle to continually assess complexity. Using metrics to determine and reduce complexity would contribute to the production of systems that are easy to understand and analyze. However, the metrics that have been defined for systems and software are directed primarily at implementation cost and maintenance effort estimation. It is not obvious that they can be extended to address performance issues.

2.1 System Complexity

As used in this report, the word "system" refers to the entire control or instrumentation system, including computer hardware, computer software, communications, instrumentation and human operators. The control system is only part of a larger system, which also includes the process being controlled, other humans and affected portions of the environment.

2.1.1 System Complexity Defined

System complexity is a measure of how difficult a system, composed of programs, computers, other hardware, natural or artificial processes, human beings, and the interconnections between all of them, is to understand and maintain. The problem of estimating system complexity is so difficult that little more than qualitative results exist. Most quantitative measures apply to software only.

System complexity metrics rely on the number of interfaces (human, software, and hardware), and the number of programs, modules, functions, and files within the system to determine complexity. Since system complexity metrics are still experimental, system evaluators tend to use heuristic measures to evaluate systems.

System complexity can be influenced by a number of factors, including the number of components (e.g., hardware and software) in the system, the allocation of functions, the number and type of interfaces between components, and the number and sizes of files to be processed. Requirements for system safety, fault tolerance, real-time processing and operator involvement also affect complexity.

2.1.2 System Complexity Evaluation

System complexity is a major problem facing today's developer. Reducing complexity during system development saves time, money, and problems once the system is placed into production. System complexity depends upon the allocation of functions the system must perform. If a single function is allocated between multiple system components, the additional data passed between the function segments will increase complexity. In any system there is a point at which lowering the number of functions per component and increasing the number of components will increase system complexity. The higher number of components requires added data channels and data flows that will increase system complexity while keeping module complexity low. System complexity can therefore be reduced by grouping similar functions to minimize the amount of data passed between modules.

Database requirements affect system complexity. As the number and types of files to be processed increase, the complexity will increase. The requirement to process a relational database or provide multiple-user access to the database produces a complexity level higher than do the requirements for a single-user access flat database file.

Requirements for system safety and fault tolerance impose additional complexity through the processing necessary to respond to faults, and the circuitry or processors necessary to compensate for single-point failures. Real-time processing requirements may add complexity by adding to the number and speed of data channels to sense and respond to system conditions, and the number of processors needed to meet response time requirements. In control systems where the goal is to minimize operator involvement and administrative procedures, the additional processors to accomplish this goal will add complexity to the computer system. Data channels to provide sensor data to these processors also add complexity. One goal of system design is to balance this added complexity against the added functionality, safety and performance of the system.

2.1.3 Complexity Attributes

Due to the scarcity of system complexity metrics, the idea of system complexity attributes has been devised. This is a common-sense approach to measuring system complexity, but has not been proven. A less precise alternative to formal complexity metrics is to identify the use of practices that have added difficulty in the past usage, are difficult to analyze and verify, or that are still topics of research and controversy. The presence of these attributes then is presumed to indicate difficulty, risky techniques or algorithms, or controversial approaches. The British Ministry of Defence (MoD) takes this stance (MoD 1991a) when it rejects certain practices that cannot be analyzed by currently available formal

methods. Other areas of potential difficulty are specifically mentioned in testing and performance analysis requirements (MoD 1991a,b). In the following discussion, a list of six complexity attributes (not exhaustive) is introduced and justifications are given for items in the list:

- Timing complexity.
- Priority complexity.
- Resource complexity.
- Connection complexity.
- Concurrency.
- Communication delay.

Some practices discouraged by MoD (1991a) are not included: floating point arithmetic and recursion, because they are algorithmic[16] instead of structural; interrupts, because interrupts are subsumed by priority and resource complexity. The concept of concurrency is used in a more restrictive sense than in MoD 1991a.

Timing complexity refers to the additional complexity incurred by scheduling sequences of events to occur at future times. In the simplest polling loop systems, a sequential loop of actions is repeated endlessly. In a more complex, scheduled system, actions may be scheduled to occur at future times unrelated to polling frequency. The introduction of future scheduling means that the system no longer responds in strict order of input, but can reorder output responses. Verification of timing is specifically required in MoD 1991a 35.5 (c), MoD 1991b 30.5.1.3, MoD 1991b 33.4.2 (c), and MoD 1991b 35.7 (c). Timing and time relations in real-time systems continue to be topics of current research (Fidge 1991). Modern real-time systems still exhibit timing errors (Riezenman 1991).

Priority complexity introduces an additional concern of precedence. If two or more actions are scheduled (or initiated by interrupt)[17] to occur at the same time, a priority scheme determines which action shall be executed first. Priority schemes introduce the additional complication of synchronization, and may reorder output responses both by time and precedence. Priority and scheduling are still hot topics in real-time system research (Sha et al 1990, Sha & Goodenough 1990).

When real-time systems become multi-tasking or multi-threaded,[18] more complexity is introduced when system resources are shared between tasks or code threads. With priority schemes and resource management, deadlocks become a potential problem. Deadlocks have been a source of trouble in many large systems, for example see Kleinrock 1978. MoD 1991b 30.5.1.3 specifically names deadlock as a subject to be investigated during performance analysis. The problem of scheduling and resource allocation in real-time systems is still a topic of research (Zhao 1987). Resource complexity is the object of one metric in the literature (Hall & Preiser 1984).

Connections in a large real-time system can be logical or physical. If the system is implemented as a number of tasks (Hoare 1978), the tasks may be connected by inter-task communications. These connections are logical and may be implemented as operating-system-mediated inter-process communications[19] in a single processor, or as some combination of logical and physical links if the tasks are running on a multiprocessor system. MoD 1991b 31.3.3 (c) requires that Static Path Analysis be performed and that software should be neither "over-modularized" nor "unduly monolithic."

---

16 Both can be replaced by other ways of doing the same thing.

17 MoD 1991a 30.1.3 prohibits interrupts which are difficult to analyze because of unplanned preemptions, a simple form of prioritization.

18 MoD 1991a 30.1.3 prohibits concurrency, in this case apparent concurrency.

19 See Preckshot and Butner 1987 for a description of such a system. The inter-process communication system (IPCS) may extend over multiple processors and may be operating-system independent.

Connection complexity is an extension of these concerns to the complication of intertask communication and cooperation, where Static Path Analysis may be difficult or impossible.

While a single-processor, multi-tasking real-time system may exhibit apparent concurrency,[20] only multiprocessor systems can exhibit true concurrency, which is the kind under consideration here. MoD 1991a 30.1.3 prohibits concurrency in the wider sense of apparent concurrency. True concurrency is defined as two or more instruction streams being executed simultaneously, which requires two or more instruction execution units (essentially, two or more CPUs). With true concurrency, it is no longer possible to assign strict time ordering to instruction streams in the real-time system, with concomitant increases in complexity in the concepts of simultaneity and synchronization. These difficulties make true concurrency, or timing in distributed systems, a current research topic (Fidge 1991).

Communication delay not only introduces additional uncertainty into intermachine timing, but significantly complicates recovery from failures. Since some delay is normal and expected, failure recovery must include waiting periods to ensure that recovery procedures are not started when there actually is no problem. MoD 1991b 30.5.1.3 (i) addresses the issue of communication delay, but not in the more difficult context of fault recovery. Current research (Muppala et al 1991) is directed toward system response and deadline performance in the presence of faults.

The design of a large real-time system involves a great many different factors. System complexity measurement is one method used to determine the proper balance among the factors. For example, a layered distributed approach to system design can achieve functional isolation, leading to improved safety and performance. The taxonomy given in the next section for system complexity should be viewed as descriptive. While a complex system is generally more difficult to analyze and test than a simple system, the complexity may be necessary to achieve important system goals. One must look at the entire problem being solved when evaluating the complexity of the system.

2.1.4 Simplified Taxonomy of Real-Time Systems

Using the complexity attributes just presented, real-time systems can be classified into groups, based upon presence or absence of attributes. Since such a classification scale is not validated against some software quality, it should be considered as being plausible only. The taxonomy proposed below is similar to the classification scheme used in biology, wherein products of later evolution are generally larger and more complex. Like biological evolutionary taxonomies, the ordering in this taxonomy may be imperfect. To validate this scheme as described in Rombach 1990, a target quality (goal) would first need to be selected—for example, the effort needed to devise, use, and analyze a performance model.[21] This effort should be accurately quantified (in man-hours, for instance) and then measured for a reasonably large number of candidate real-time systems. This provides a sample ensemble consisting of ordered pairs, the real-time system taxonomy level, and the effort required to estimate its performance. The ensemble is examined for correlation between level and effort. The hypothesis that taxonomy level is a predictor of performance estimation effort is conditionally sustained if the correlation is high enough, and falsified if there is no correlation.

An assumption is made in this taxonomy that real-time systems contain loops, and may optionally deal with preemptive events. This is true in the broad sense that any finite discrete system eventually repeats itself, but is also true in that real-time systems are often cast as repetitive tasks. Using the

---

[20] Brinch Hansen 1975, for example, defines concurrent programming constructs which are actually executed serially on a single processor. Apparent concurrency is concerned with synchronization and code segments called critical sections.

[21] Devising the performance model may be the simplest task. Deciding what simulated loads to apply and interpreting the results may consume a large part of the effort.

attributes sidesteps the issue of code size. The results of Henry and Kafura (1981) in particular, and the general uncertainty of Source Lines of Code (SLOC) or size as a complexity measure, seem to indicate that structural measures are more reliable indicators of complexity. Relying upon software structure alone for classification ignores a number of knotty practical problems such as visibility into system internals and the effects of hardware on software operation. In the following taxonomy, hardware is considered in a very broad sense,[22] and operating systems are not explicitly separated from other code. Where operating systems are used,[23] the system should be classified as the more complex of either the operating system, the real-time application code, or the combination of the two.

2.1.4.1 Single-Processor Systems

In most cases it is simple to decide the number of processors in a system. However, coprocessors and auxiliary processors such as vector accelerators add confusion. Systems with closely coupled coprocessors such as math coprocessors are considered to be single CPU systems. Smart interfaces, such as Ethernet interfaces, Small Computer System Interface (SCSI) controllers,[24] and some video accelerators still do not constitute a multiprocessor system. However, a separately programmed auxiliary processor which executes almost-general-purpose code in parallel for significant time periods probably qualifies as an additional CPU.

2.1.4.1.1 Polling Loop (Level 0)

The simplest of all real-time system organizations is the polling loop. This system exhibits no complexity attributes, and is a single loop with sequential execution and (almost) no interrupts. Branches are allowed to select alternative actions depending upon input values. A single timer interrupt is sometimes allowed as long as it is used only to update a clock or time variable. The polling loop executes continuously and endlessly, and the loop time is determined only by the amount of work performed during each iteration.

In terms of total number of systems extant, polling loops and scheduled polling loops may be the most popular types of real-time systems by a wide margin. Simple Programmable Logic Controllers (PLCs), embedded digital controllers, and simple, ad hoc laboratory control systems all benefit from the simplicity (and hence reliability) of the polling loop.

In spite of the simplicity of the polling loop, it is still possible to ruin its reliability through poor design. Poorly designed input/output interfaces can stall the loop by causing it to wait for something that never happens.

The scheduled polling loop is still a level 0 organization. Simple polling loops run at whatever speed the loop load allows. Some applications, such as digital signal processing, digital filters, proportional-integral-derivative (PID) controllers, and other applications of digital techniques to continuous-time systems, require precisely timed input/output cycles. The polling loop can be extended to cover these applications by starting each loop cycle at precisely timed intervals. The scheduled polling loop does not exhibit the timing complexity attribute because outputs still occur in the order of inputs. In order to perform correctly, loop time must not exceed the precise timing interval, but this can be determined by static timing analysis.

---

22 For far more detail on hardware, see, for instance, Savitzky 1985 chapter 2.

23 We consider run-time kernels in embedded systems to be operating systems.

24 See Savitzky 1985, chapter 2.

2.1.4.1.2 Schedule Loop (Level 1)

Unlike the polling loop, the schedule loop does not visit a fixed list of inputs and tasks,[25] but instead continually polls a dynamic schedule. Things-to-do are placed on the schedule either repetitively, by themselves, or by other, previously executed things-to-do. Inputs can be polled at scheduled times, so the schedule loop reduces to the scheduled polling loop as a degenerate form. The schedule loop exhibits the timing complexity marker, but in this instance timing correctness may only be verifiable dynamically, unless strict rules are followed for self-rescheduling or cross-scheduling. Schedule loops can be used for a surprising number of real-time applications, including flight control systems and mission supervisory control systems on spacecraft.

2.1.4.1.3 Preemptive Single Task (Level 2)

Both timing and priority complexity attributes are exhibited by preemptive single task real-time systems (often called "event-driven systems"). Any kind of preemption introduces a priority scheme, whether explicit or implicit. Preemption introduces unplanned delays in the preempted program in order to service higher-priority demands with minimal delay. The typical preemptive real-time system services interrupts while running a real-time application (such as a schedule loop) as the non-interrupt single task. Since the application may be in a delicate state at the time an interrupt occurs, these systems often have methods to synchronize the incoming data from the interrupt with an opportune point in the application code.

Although not formally called tasks, interrupt service routines actually execute in different contexts from the real-time application, and thus fulfill at least some of the requirements for being called tasks. When completion routines are executed as a result of interrupts, the distinction blurs further, since completion routines,[26] signal handlers,[27] and asynchronous trap routines[28] are usually considered a form of light-weight task. Nonetheless, we consider systems with such features as single-task systems, mostly because the issue of resource management is not raised. In single-task systems, all resources are available to the real-time application task, and there is no resource contention.

2.1.4.1.4 Preemptive Multitasking (Level 3)

Timing, priority, and resource complexity attributes are displayed by preemptive multitasking single-processor real-time systems. If the tasks communicate, logical connection complexity is present as well. The multitasking preemptive system is similar to the single-task system, but now several tasks (which appear to be separate code threads in different contexts) may contend for execution time and for system resources. This level of complexity is the first level at which it is common to see subtle errors due to interactions between timing, priority, and resource allocation. Deadlock is the most feared problem, but performance deficits due to resource contention may be more common and harder to detect.

Foreground/background systems are common examples of small, preemptive multitasking single-user systems,[29] but almost any combination can be found in embedded, non-interactive real-time systems.

---

25 Here task is defined as something to do, as opposed to the definition found in the glossary.

26 Digital Equipment Corporation (DEC) RT-11 operating system synchronization mechanism.

27 UNIX operating system synchronization mechanism. Its mention here is not meant to imply that UNIX is a real-time operating system.

28 DEC VMS operating system synchronization mechanism.

29 Usually a small computer system which performs real-time chores as the foreground task while attending to its master's voice in the background.

2.1.4.2 *Multiprocessor Systems*

When a single processor can no longer support the required computational load, or when data and input/output devices are physically separated, additional CPUs are added to increase total processing speed or to manage data and devices in local areas. Many ways exist to connect CPUs together; they may be roughly divided into closely coupled versus loosely coupled systems. The principle for separating systems is that closely coupled systems communicate between CPUs almost as fast as the CPUs can process, thereby making it practical to subdivide processing jobs rather finely and still retain an advantage. Closely coupled systems tend to be coupled by shared memory mechanisms, and loosely coupled systems usually are connected by serial lines or networks such as Ethernet or fiber ring. There are exceptions to this,[30] just as there are exceptions to the generalization that closely coupled systems are usually used to increase processing power, while loosely coupled systems are used to manage physical separation. Further discussion of specific examples of networks and data communications is left to another report.

To be considered a multiprocessor system, code running on one processor must be dependent upon code running on another. Some combination of resource or connection complexity must tie concurrently executing code together (see Lehman and Belady 1985 for a discussion of what constitutes a system), or the software must exhibit interdependencies through the physical system being controlled[31] (e.g., actions by one software module cause physical system changes upon which another software module depends for proper operation). Independent real-time systems that do not affect each other should be considered at their individual complexity attribute levels.

2.1.4.2.1 *Closely Coupled Systems (Level 4)*

Closely coupled systems are usually in the same room or in the same electronics rack, and are usually connected by wide (many conductor) memory buses.[32] A useful discussion with examples of actual commercial shared-memory or shared-bus multicomputer systems can be found in Weitzman 1980, chapters 2 and 4. Because several connected processors may attempt simultaneous access to shared memory, there must be at least simple hardware arbitration to prevent inconsistent data from being written to shared memory. Often there is very little more than this, and software bears the burden for higher-level coordination. However, some processors, such as the INTEL 432 and the INTEL 960, have substantial functions built into their hardware to support system-level shared memory management, and also to support shared-memory, multiprocessor operating systems.

Symmetric Systems. Less complex among closely coupled systems are symmetric multiprocessor systems, where both hardware and software[33] are replicated as more processors are added. These systems exhibit at least timing, connection, and concurrency complexity, and are likely to exhibit priority and resource complexity as well. Performance models may have to include models for processor scheduling. In systems with replicated tasks but different data, performance models are simplified, since the model can be replicated as well.

Asymmetric Systems. If a system is asymmetric, some hardware and software differ depending upon which CPU is under consideration, which increases the complexity of the performance model. These

---

30 For example, Transputers are interconnected by very fast serial links, not shared memory. Transputer systems are considered closely coupled, and are commonly used to increase total available processing speed.

31 This is an unfortunate situation, because software correctness becomes dependent upon correctly understanding details of the physical system.

32 Except, of course, Transputers.

33 There may, in fact, be a single distributed operating system and a set of distributed application tasks which run on whatever processors are ready.

systems exhibit at least timing, connection, and concurrency complexity, and are likely to exhibit priority and resource complexity as well.

2.1.4.2.2 Loosely Coupled Systems (Level 5)

The component parts of loosely coupled systems are often separated by hundreds of meters, and are usually connected by some sort of network technology. Communication delay is a significant factor, and is included *ab initio* in the design process. These systems exhibit at least timing, connection, concurrency, and communication delay complexity attributes, and are likely to exhibit priority and resource attributes as well. While it is possible to have a single "network" operating system, typical loosely coupled multiprocessors have separate operating systems running on each CPU.

Symmetric Systems. These systems have identical nodes emplaced in a symmetric communication network. The software running on each node is the same, but the data or sensors may be different, requiring communication to obtain non-local data. Performance modeling is simplified because each node is replicated and embedded in the performance model for network communications.

Asymmetric Systems. If a system is asymmetric, then some hardware and software are different, depending upon which CPU is under consideration. This increases the complexity of the performance model, which may include different models for each node and an asymmetric communications model.

2.1.4.2.3 Hybrid (Level 6)

Hybrid systems consist of processors of various design connected by networks in clusters of shared-memory multiprocessors. Such systems have been used successfully in large control systems (Wyman et al 1983). These systems are the most complex that one can encounter, and usually exhibit all complexity attributes.

2.2 Software Complexity

2.2.1 Software Complexity Defined

In contrast to system complexity, software complexity is an indicator of how difficult a program, module, or the functions performed are to understand or modify. It is generally agreed that a software module must be at least as complex as the function it is supposed to provide. Quite often, software is more complex than this minimum, and one objective of measuring software complexity is to estimate both the minimal and actual complexity. Historically, producers of software have had the most motivation to attempt measurements of complexity and other software attributes that have economic impact.

The science of measuring software is called "software metrics." Metrics exist for size, cost, style, difficulty, structure, errors, and reliability, to name several. Aspects of software complexity have been discussed by several authors in widely varying ways, because the concept of complexity is inherently disordered. There is probably no finite set of numbers which can describe even a partial ordering from simple to complex. The current practice in devising software metrics, nicely summarized by Rombach in the following paraphrase, provides both an approach to selecting aspects of complexity to measure and a framework for understanding metric research.

0. Define a goal.[34]
1. Model the quality of interest and quantify it into direct measures.

[34] Step 0 was abstracted from Rombach's goal/question/metric (GQM) paradigm description as being a reasonable preamble to the steps described for the author's Distos/Incas experiment.

2. Model the product complexity in a way that lets you identify all the aspects that may affect the quality of interest.
3. Explicitly state your hypothesis about the effect of product complexity on the quality of interest.
4. Plan and perform an appropriate experiment or case study, including the collection and validation of the prescribed data.
5. Analyze the data and validate the hypothesis.
6. Assess the just-completed experimental validation and, if necessary, prepare for future experimental validations by refining the quality and product-complexity models, your hypotheses, the experiment itself, and the procedures used for data collection, validation, and analysis.

(Rombach 1990)

Defining a goal may seem an obvious precursor to devising a metric, but it may not be an obvious precursor to choosing a metric from a catalog of existing metrics. Metrics are validated by experimentally determining the correlation between the metric value and the particular aspect or quality of software which the metric presumably affects. Metrics in the literature have been validated against such software qualities as development effort, debug time, errors per line of code, difficulty, or changes per software module during maintenance. These qualities often have narrow scope and it may be difficult to conclude that they are related to other goals and purposes.

A software developer may wish to measure certain types of complexity in a design-in-progress for the purpose of moving effort or scrutiny to where it is most needed. A software tester may use metrics to determine where to direct the most testing effort. The NRC may wish to require certain maximum levels of complexity for safety-critical software. For this it would need a measure or attributes to describe unacceptable complexity. Reviewing a system design after it is complete, the NRC will be concerned with estimating the effort required to validate a design, estimating the effort required to predict system performance, or estimating the hazard level of a design. In these cases, metrics would be needed that were correlated with validation effort, the effort to implement and use a performance model, or the number of hazardous errors contained in either the design or its implementation. As a practical matter, any metric chosen or devised would also have to be calculable with the data available at the time it was needed.

2.2.2 Software Complexity Metrics

A large number of complexity metrics for software have been invented. Most of these are aimed at reducing the cost of developing or maintaining software, or at increasing its correctness. It is not obvious that measures relating to such aspects can be directly extended to issues relating to performance, reliability or safety. Thirteen of the more popular complexity metrics are described on the next few pages.

2.2.2.1 *Source Lines of Code (SLOC) Metric*

One of the metrics in software development has been delivered SLOC; the intent has been to measure effort required to complete a software project. This metric was chosen mostly because it was easy to determine and because it seemed reasonable that effort was proportional to the amount of code written. Source lines of code has had indifferent success either in predicting software effort or in recording expended effort (Brooks 1975). "Lehman and Belady (1985), in attempting to characterize large systems, reject the simple and perhaps intuitive notion that largeness is simply proportional to the number of instructions, lines of code, or modules comprising a program" (Burns & Wellings 1990). In

another experiment, debugging effort was shown to be independent of SLOC (Gorla et al 1990). One reason given for imprecision in SLOC is that some programmers are prolix while others are terse (Boehm et al 1984). Regardless, SLOC continues to be used either because it is easy to compute or because it is used as a yardstick against which other metrics are compared.

2.2.2.2 *McCabe's Complexity Metrics*

Because of the poor success of SLOC or size metrics at quantifying difficulty or expected effort, other metrics more related to debugging, implementation, or maintenance effort have been proposed. Rather than code size, these metrics attempt to measure some aspect of software complexity and use it to predict or quantify the target software quality of interest. McCabe (1976) suggested a graph-theoretic metric which measures complexity in terms of linearly independent paths through the code. He calls this "cyclomatic" complexity. A second complexity measure, which he calls "essential" complexity, measures how much the complexity of a program module can be reduced by conversion of single-entry/single-exit (SE/SE) subgraphs to vertices (i.e., subroutine calls). Essential complexity is the cyclomatic complexity with all SE/SE subgraphs converted to subroutine calls. If the essential and cyclomatic complexity are equal, the module's complexity cannot be reduced. If essential complexity is less, then the cyclomatic complexity can be reduced. McCabe suggests that a complexity greater than 10 (McCabe & Butler 1989) is difficult for programmers to comprehend, and will therefore be difficult to develop and maintain. Part of the motivation for this metric was to improve the testability of routines by putting an arbitrary limit on size. This metric requires access to code or to a graphical representation of the module design,[35] and is useful at the module or subroutine level. McCabe presents examples of use and gives anecdotal reports of metric usage. There is no formal validation, but others (to be described) have used McCabe's metric in validation experiments. McCabe's metrics are being used by some software developers to help reduce control flow complexity.

2.2.2.3 *Software Science Metrics*

Halstead (1977) introduced "software science" metrics based upon counting operator and operand tokens in source code. Various arithmetic combinations of operator, unique operator, operand, and unique operand counts are used as measures of length (N), volume (V), level (L),[36] difficulty (D),[37] and effort (E).[38] Basili et al (1983) examined the correlations between SLOC, Halstead's, and McCabe's metrics with coding effort and number of errors for seven projects and 1,794 Fortran modules at NASA/Goddard Space Flight Center. This represents an independent attempt to validate these metrics against two software qualities: coding effort and number of errors. None of the metrics tested consistently showed more than moderate correlations with effort or errors. The Basili paper concludes:

1. None of the metrics examined seem to manifest a satisfactory explanation of effort spent developing software or the errors incurred during that process.
2. Neither Software Science's E metric, cyclomatic complexity nor source lines of code relates convincingly better with effort than the others.
3. The strongest effort correlations are derived when modules obtained from individual programmers or certain validated projects are considered.
4. The majority of the effort correlations increase with the more reliable data.

---

[35] See Henry and Selig 1990 for an example of applying this metric prior to coding.

[36] The theoretical minimum number of bits required to code divided by the actual bits required to code.

[37] 1/L, supposedly the difficulty required to code the algorithm.

[38] V/L, supposedly the effort required to understand the implementation of the algorithm.

5. The number of revisions appears to correlate with development errors better than either Software Science's B metric, E metric, cyclomatic complexity or source lines of code.
6. Although some of the Software Science metrics have size-dependent properties with their estimators, the metric family seems to possess reasonable internal consistency.

(Basili et al 1983)

2.2.2.4 *Information Flow Metric*

More convincing correlations were reported by Henry and Kafura (1981) using a metric based on "information flow." This metric measures complexity in terms of data connections "fanning in" and "fanning out" from modules. The Henry metric represents a significantly different approach from the Halstead–McCabe metrics. It is a "structure" metric, as opposed to a "code" metric (Henry and Selig 1990). As a validation for the metric, Henry used the UNIX operating system[39] as the study subject, and the number of changes per module during life cycle maintenance phase as the quality to be predicted. Henry reports an impressive 0.98 correlation between percent-of-modules-changed versus a metric defined as:

$$(\text{fan-in} \times \text{fan-out})^2$$

Notably, she achieved better correlations when a size-dependent term was removed from the expression.

In a later experiment, Henry examined the performance of the information flow metric, SLOC, cyclomatic complexity, and the Halstead Software Science metrics on programs written by senior-level software engineering students at Virginia Polytechnic Institute and the University of Wisconsin at LaCrosse (Henry and Selig 1990). Using students as sources of research material is controversial because, as critics point out, the academic environment differs markedly from the commercial and professional environment. In particular, academic projects are often small, simple, and relatively well constrained as far as what problems can be addressed. In this case, Henry reports that the projects were large (3000–8000 SLOC), the specifications were not provided by the professor, and the students had no access to experimental statistical data. The metrics were applied in three design refinement levels to 276, 283, and 422 software modules respectively. The numbers and distribution of student programmers involved were not mentioned. The results show a wide range (a factor of two) between predicted and actual complexity for the information flow metric at 95% confidence level.

2.2.2.5 *Style Metric*

An example of a "style" metric is provided by Gorla et al (1990) in an attempt to predict debugging effort from style attributes (lexical complexity) in Cobol programs. These attributes include such things as percent comment lines, percent blank lines, number of goto statements, indentation, and the like. As controls, SLOC and cyclomatic complexity were included. The validation attempt used 311 student programs collected from five intermediate Cobol programming classes as raw material. Gorla reported that nine of fourteen metrics had values which correlated with minimal debug time (e.g., variable names between 12 and 16 characters long correlated with short debug times). SLOC and cyclomatic complexity showed no significant correlation with any of three debug measures.

---

[39] "The third reason for selecting UNIX is the fact that UNIX software was designed for users and not as a toy or experimental system. In particular, it was not written to illustrate or defend a favored design methodology" (Henry and Kafura 1981).

2.2.2.6 *Information Theory*

Sahal (1976) described an information-theory-based system complexity metric which measured complexity based on four behavioral components, viz. organized, unorganized, short-term, and long-term behavior. These roughly correspond to deterministic, non-deterministic, short-term time series, and long-term patterns of behavior. Sahal presented an application of his model to the evolution of the technology of piston-engined, carrier-based aircraft. The sample size is one, which makes the interpretation of the claimed correlations unconvincing. The idea of a system complexity metric with a mathematical basis is attractive, but a significant amount of research would be required to apply this metric to software systems.

2.2.2.7 *Design Complexity*

Working from software structure charts, McCabe and Butler (1989) extended their module cyclomatic complexity metric to graphs that describe how code modules work together, rather than how individual modules work. "Module design complexity" is a per-module complexity measure that describes the number of unique basis paths by which a module decides to call subordinate modules. "Design complexity" is the sum of the individual module design complexities taken over the entire software structure chart. "Integration complexity," the third metric of this group, is a measure of the integration tests required to qualify the design. Like the original McCabe cyclomatic complexity metric described above, no validation results are presented in McCabe and Butler 1989.

2.2.2.8 *Graph-Theoretic Complexity*

Another attempt to apply McCabe's cyclomatic complexity measure to networks of modules resulted in the "graph-theoretic complexity for architecture" metrics (Hall and Preiser 1984, IEEE 1988b). The intent of this approach was to devise complexity metrics that accommodated concurrent systems modeled as networks of potentially concurrently executing tasks. These complexity metrics require system representation as a strongly connected graph, with the nodes being software modules (tasks) and the edges being transfers of control (messages). "Static complexity" is then defined as:

$$C = \text{edges} - \text{nodes} + 1.$$

"Generalized static complexity" combines a complexity estimate for program invocation with resource allocation complexity, summed over the elements of the network graph. This metric attempts to measure the complexity associated with resource management, synchronization, and communication. The third metric, "dynamic complexity," is static complexity evaluated for actual module invocations over periods of time. It is intended to account for the effects of actual frequency of invocation and module interruptions. Published reports of usage of these three metrics in the open literature are limited. Generalized static complexity and dynamic complexity have not been reported in operational use (IEEE 1988b).

2.2.2.9 *Design Structure Metric*

The "design structure" metric (IEEE 1988a,b) produces a complexity number that is the weighted sum of six "derivatives" calculated from seven measurement primitives. The measurement primitives are counts of modules or database elements having certain qualities, rather than internal details of the modules themselves. The metric attempts to quantify dependence on input/output, dependence on stored state, database size, database compartmentalization, and excess module entries/exits. Reported experience for this metric is limited and not easily accessible.

2.2.2.10 *Function Points Metric*

Albrecht and Gaffney (1983) present a "function points" metric that uses external system stimuli and functional requirements as its inputs. As an alternative to direct size estimation, function points attempt to characterize software projects by input/output connections and the difficulty of functions performed. The metric is therefore applicable at an earlier stage of the design process, when little or no coding has been done. Albrecht reports correlations over 0.9 with "work-hours," a measure of time required to implement an application. However, relative standard deviations for Albrecht's metrics are in the range 0.4 to 0.8, which suggests a typical relative error of a factor of two. Function points are computed using "external input types," "external output types," "logical internal file types," "external interface file types," and "external inquiry types." These items appear to be closely related to business data processing practices, as do the detailed rules for counting and combining them.

2.2.2.11 *COCOMO*

The COnstructive COst MOdel (COCOMO) model (Boehm 1981) for estimating software project effort and expected project time is an empirical model relating SLOC with expected man months and total months required to complete a project. The basic COCOMO model uses two simple equations to compute man months and project time:

$$MM = K\,(SLOC)^P$$
$$TIME = R\,(MM)^q$$

K, p, q, and R are functions of project team type. How the original estimate of SLOC is arrived at is never explained, but in early project stages SLOC estimates are notoriously bad, and the exponent p is greater than one. The accuracy of the basic model is claimed to be worse than a factor of two about 40% of the time, based on validation efforts on a database of 63 projects. The detailed COCOMO model is a weighted-SLOC model with 15 cost-driver weights. This is a complex model requiring a lot of subjective judgments. The accuracy claimed for the detailed model is no more than 20% error from actual development effort 58% of the time.

2.2.2.12 *Cohesion*

The cohesion metric (Yourdon and Constantine 1979) is an indication of how well a module conforms to a single function. Single-function modules are usually less complex and more understandable than larger multifunction modules. Modules with strong cohesion perform only one function. When design changes are needed or errors occur, these modules can easily be understood and changed. A module that performs a group of related functions in a logical order is more complex than modules that perform one function. A randomly cohesive module performs related functions in a random order. Using the same module to update several databases and retrieve data from them is an example of random cohesion. Modules with weak cohesion process unrelated functions in any order and should be avoided.

2.2.2.13 *Coupling*

Coupling (Yourdon and Constantine 1979) describes how well a module interfaces with its units and how well the units interface with each other. The coupling metric measures how well modules and units pass data. The best type of coupling is data coupling; only the required data is passed from unit to unit. Stamp-coupled units pass entire data structures instead of just the necessary data. Externally coupled units share the same global data items, but not the entire global data base. When two or more units share the common database they are common-coupled. Common coupling allows any unit to access and modify common or global data, adding to complexity by expanding the number of places that data can

be accessed. Lastly, content coupling means that one unit can directly reference data inside another unit. Content coupling is the least desirable form of coupling.

3. Scalability

An important question is, "At what point does a small, fast, real-time system, when extended to the large, become a big, slow failure?" The property that describes the ease of extension of small to large is termed "scalability." Scalability includes both the extension of actual small systems to large systems and the application of small system techniques inappropriately to the large system domain (thinking small). Unfortunately, knowing how to scale real-time systems is more a matter of experience than theory.

"Only when we have built a similar system before is it easy to determine the requirements in advance" (Parnas 1985).

Most of what is known has been gained by experience, not forethought, because scaling difficulties tend to be specification oversights, not software development method errors. Some techniques have been described for anticipating system scaling (Parnas 1979) but retrofit to existing systems is problematic. Some issues in system scaling, such as capacity and resource usage, are susceptible to quantitative modeling, and performance is usually predictable, provided that realistic load patterns are used to drive the models. Other areas are more difficult to predict and may produce unpleasant surprises.

3.1 Synchronization

The first and most obvious effect of replicating CPUs and connecting them together is that time is no longer the same at different points within the system. At the very least, complete time ordering degenerates to partial orderings of events occurring within each CPU. Current research in real-time systems focuses on how to ensure that supersets of such partial orderings in real-time distributed systems are causal (Fidge 1991). The state of practice is not reassuring, and synchronization and time ordering within distributed real-time systems is an area of continuing research. Real-time system developers continue to make timing and synchronization errors.

Timing errors show up as incorrect (or mismatched) times associated with events or objects, or as incorrect ordering of events, data, or messages. The range of errors this can cause is enormous. The system can drop data because of apparently incorrect time stamps, behave erratically because of incorrect order, deadlock waiting for something out of order, or freeze completely because fault detection circuitry or software stops the system on synchronization failure.

Arguably the most famous public example of a system freeze may be the Space Shuttle synchronization error (between four redundant CPUs) that caused a delay during the first Shuttle launch attempt (Lombardo and Mokhoff 1981). This was caused by the elementary error of using different clocks for different processes, resulting in a time skew between processes. This would not have been detected (and may not have been significant) in a single processor system, but was crucial in a cross-checking multiprocessor system. Fortunately, this was resolved with no injuries except to the pride of IBM Federal Systems Division. That it could happen to such a large and experienced organization is indicative of the difficulty of synchronizing distributed systems.

3.2 Precision

Precision refers to the ability of a variable to represent some quantity important to the functioning of a system. Precision limitations can exacerbate the effects of poor design, or they can engender sudden, unexpected difficulty when enlarging some system dimension, typically an address or name space.

The designers of the PDP-11[40] have reported that they consider their worst design error to have been limiting the address representation of the machine to 16 bits, which makes the maximum direct address 65,536 (64 Kbytes). Later, Intel repeated the error, more creatively, in their 80x86 architecture microprocessors, and Microsoft enshrined it in the MS-DOS operating system, which, even after five major versions, is limited by the well-known "640K barrier." The architectural limitation is so pervasive that many compilers for 80x86 computers do not support data structures larger than 64 Kbytes.

The most recent public failure due to time precision was the failure of some Patriot missiles in the Gulf War. Initial reports cited an "imprecise timing calculation" that resulted in a cumulative error of 360 milliseconds after 100 hours of Patriot battery operation (Riezenman 1991), which caused the system to drop an incoming Scud missile from its target list. A later GAO report more accurately stated that the error resulted from an integer-to-floating-point conversion that had only 24 bits of precision. Since the clock hardware had integer registers, the error grew proportionally to the time the system had been running without a restart. This was a system specification error caused by a lamentable lack of foresight on the part of the system specifiers, who failed to anticipate system operation expanded along the time axis. Real human operators sometimes forget to compensate for built-in system deficiencies, as was the case in the Gulf.

Precision limitations may show up as a frank failure to adapt to larger scale applications, but unfortunately the more usual symptoms are progressively more contorted attempts to "band-aid" the system into operation, with decreasing reliability and performance. The effects are insidious and often are not acknowledged until a failure or severe performance bottleneck appears under load or fault recovery.

3.3 Capacity

Systems sometimes run into severe performance problems when the capacity of an innocuous component proves to be the system bottleneck. The OS-32 operating system[41] provides a classic example. This system did not spool operator error messages, but instead held the messages and the tasks sending the messages hostage to the speed of the operator's console. Naturally, when serious problems occurred, many tasks wrote messages to the operator's console, and the "real-time" system slowed to the speed of a mechanical teletype machine. The modern-day equivalent is the overloaded logger.

Network links are a significant source of capacity problems. Often, when a distributed system is expanded, the traffic between parts of the system is imperfectly understood, especially under fault conditions. This is exacerbated by the tendency of many to assume that raw network data rate is available to the tasks running in computers attached to the network. In fact, due to protocol overhead and operating system delays, the usual effective data rate for a computer attached to an unloaded Ethernet (1.25 Mbyte raw data rate) is 50–600 Kbytes. This ratio is typical of many networks.

---

[40] A Digital Equipment Corporation computer circa 1972.

[41] A multitasking operating system on Perkin-Elmer computers circa 1983.

Capacity problems occur because of insufficient analysis of data rates within the system and their effects on system devices and datalinks. When the system is small, potential capacity problems do not surface, because in absolute terms the data load does not stress the system, regardless of architecture. As the system is enlarged, more physical dataflows are then overlaid on datalinks and devices. The difference between the logical view and its physical implementation obscures pertinent detail. However, with the exception of certain effects discussed under the next heading, capacity expansion can be dealt with quantitatively by using performance models to predict the effects of increased load or different load patterns.

3.4 Load-Correlated Load

Linear estimates of capacity scaling can be vitiated by load derivatives that are positive functions of load itself. One area where this is likely to occur is in fault recovery protocols for transient, operationally expected faults. This can be particularly difficult to detect before the system is developed, because detection may require detailed analysis of fault handling algorithms and their interactions with architecture and computing environment. There also may be no sign of impending failure until a system actually becomes large and loaded. In small systems the rate of fault occurrence may be insufficient to incur much additional load during infrequent fault recovery. Expanding the system, however, may increase both the frequency of fault occurrence and the time required to recover from each fault. Now fault-handling protocols that are triggered during load-correlated system activities may cause sudden system collapse above critical load levels. In effect, efforts to handle and recover from faults place additional load on the system, which causes further attempts to handle faults. At some load level, this is regenerative and system throughput drops almost to zero.

Early network communications research provides an illustrative example. Packet-switched carrier sense multiple-access (CSMA) data links have an unstable[42] throughput versus offered-load curve (Kleinrock 1975). This is because the error recovery procedure (simple retransmission) can multiply network load[43] at high packet collision frequencies. The Ethernet (Metcalfe and Boggs 1976) is stable under high load because the collision recovery procedures defer load rather than increase it (Shoch 1980).

The effect of human operators as impromptu fault recovery mechanisms should not be neglected. Operator actions that are intended to gain more information may place more load on the system, and operator impatience correlates well with slow system response, which in turn correlates highly with system load.

3.5 Intermediate Resource Contention[44]

Intermediate resources are resources required because an intermediary, often hidden, is placed in a logical dataflow when a system is expanded. They include buffers to store and forward data packets, disk buffers for database access, and locks, semaphores, and other operating system resources. At the worst, contention can result in deadlock (see Kleinrock 1978 for data communications examples), but,

---

42 As offered load increases, throughput peaks and then drops.

43 The packet is transmitted to completion during the collision, and then transmitted again during retransmission, at least doubling the effort required to send the packet.

44 For more discussion of resources in real-time systems see Burns and Wellings 1990, Chapter 11. The general subject of resource management is more extensive than can be covered here.

more insidiously, can result in performance degradation. While the previous comments on capacity can also be construed as a resource management problem, it is possible to have datalinks of adequate capacity whose performance is reduced by insufficient buffer allocation in one or more nodes. The condition, of course, only shows up during heavily loaded operation.

Resource problems occur because of insufficient analysis of resource usage within the system and, in extreme cases, because of total ignorance of resource usage. When the system is small, resource problems do not surface because very few resources are used. As the system is enlarged and more intermediaries are introduced, hidden resource choke points may be introduced as well. The problem is similar to the problem discussed under capacity, and avoidance techniques are much the same. Resource models can be included in capacity models, and quantitative estimates of resource usage can be obtained for different load patterns.

3.6 Databases

Whether explicitly recognized or not, each distributed system has an associated database. At the minimum, this database consists of embedded assumptions and data that are compiled into or entered into software that is running on nodes of the system. At the other end of the spectrum, data may reside in a formally defined database that is controlled by a distributed database manager. The minimal approach suffers from configuration management problems; it may be impossible to verify just what data is distributed throughout the system, and changing the system may be a nightmare. The formal database approach is flexible, but sometimes at the expense of performance.

An important class of systems, industrial process control systems, has at least two databases for each example of the class. One consists of associations between signal or point names and the actual sensors which produce data. Conversion factors and procedures may also be associated with the names, and the form of this data may be PLC programs distributed in various PLCs that make up the periphery of the system. The other database consists of operator displays and associations of signal names with locations in the display graphics. The two databases may be merged into one, depending upon implementation, and there probably are additional databases not mentioned here.

Database managers present logical views of data, regardless of the actual physical storage layout of the data. Because of this, database access time is a function of database size and the number of probes required to resolve indirections and to access physically separate locations. In a distributed system, a single query may generate a flurry of distributed queries as the datum or its components are located and retrieved. This can multiply the use of intermediate resources and datalink capacity to the point that, in large systems, the performance deficit may be fatal.

3.7 Correlated Stimuli

For economic reasons, in some distributed systems interconnections are undersized relative to the number of sensors available and amount of data potentially producible. To avoid overload, data is filtered at the source and only "meaningful" or "important" data is actually transmitted. This practice is justified and datalinks are sized by assuming that meaningful data can be described by a Markovian probability model and arrive at the boundary of the real-time system with exponentially distributed interarrival times. This assumption is made in the literature (see Kleinrock on queuing theory, for example) because it is mathematically tractable. Often, nothing could be further from the truth.

A real-time system undergoing an excursion often experiences large numbers of rapidly changing process variables that produce a large number of data correlated in time with the excursion. In addition, increased operator activity is directly correlated with lack of system response, which occurs when the system is overloaded. In systems which have variable data rates and which are undersized with respect to the maximum possible data rate, correlated data overloads should always be anticipated and the possibility carefully investigated.

4. Conclusions and Recommendations

Many factors must be combined to predict the safety, complexity and reliability of the system. The three major factors investigated in this report are system complexity, software complexity, and scalability. Software has a significant effect on the system performance because it is complex and difficult to verify.

4.1 System Complexity

No industry-accepted system complexity measure has been identified, and there is very little research in this area. A possible first alternative to firm system metrics is to make a preliminary assessment of system complexity using the complexity attributes presented in this report, or other, similar indicators. The preliminary assessment provides an idea of just how much trouble and effort to expect in determining system safety and performance. For safety-critical software, a complexity attribute level greater than one indicates possibly hazardous practices. For high-performance, real-time control systems, a level greater than two indicates that there may be significant scheduling, resource, and coordination issues to be examined. A level greater than three indicates that there may be additional synchronization and performance issues which complicate testing and performance analysis. For any real-time system, a level of four or greater indicates that scaling issues should be closely considered. As remarked earlier in this report, complexity attributes have not been validated, and the previous advice must be considered to be based on an experienced estimate only.

4.2 Software Complexity

Complexity and quality metrics have been used to guide software development, predict modification difficulty, predict testing difficulty, estimate module modification rates during maintenance, and predict the number of remaining errors. The assumption underlying these efforts is that more complex modules are harder to understand, harder to modify successfully, and more error-prone. Before software implementation, this is taken to mean that high levels of measured complexity indicate that portions of the design should be re-engineered. After implementation it is taken to mean that re-design may be necessary, or higher maintenance effort and higher levels of remaining errors may be expected.

Attempts to validate the various complexity metrics against software qualities of interest have had variable results, with one important exception, fully supported by IEEE 1988b guidelines. That result is that the success of metric usage is extremely sensitive to measurement environment. Accurate software measurement is not the work of an afternoon or even of a few weeks, but must be embedded in a

long-term program in which the metrics can be calibrated to the environment and to the people in the environment.

Another observation made about many of the metrics is that detailed application of the metric is tedious and error-prone. This is particularly true about code metrics, which require decomposition of source code into measurement "primitives." Many authors, as well as IEEE 1988b, recommend automation using source code scanner programs to reduce or eliminate tedium and errors.

The most effective way to use software complexity metrics is to begin using them before software design begins. Complexity metrics (and other metrics not covered in this report) should be a part of a software management program from the start. During design and implementation, accumulating statistics serves to calibrate metric values against the design organization. After implementation is complete, sufficient data is available to compute variances and confidence levels for complexity values of the finished product. The Halstead, McCabe, or Henry metrics are recommended because they require fewer subjective judgments and have been automated successfully. Many references use a McCabe cyclomatic complexity of less than 10 as indicating an acceptable or good complexity. McCabe reaffirms this in McCabe and Butler 1989.

Even if a solidly managed metric program is in place, caution is advised in interpreting resulting metric values. No existing metric has been thoroughly tested with complex real-time systems, and few metrics, if any, have been tested on multi-program systems. Assigning causes to high correlations is a very chancy endeavor. For example, the Henry information flow metric was validated against higher incidence of module changes during maintenance phase. Does this mean that high information flow complexity means higher error incidence? Consider that high information flow complexity indicates that a module has a greater number of connections with system data. This may mean that the module has a greater chance of being affected by an error (and thus may require modification), but not that the error actually resides within the module. Using a metric outside its range of validation may result in incorrect conclusions.

Some limited use can be made of uncalibrated code or structure metrics. Unfortunately, the most significant results these metrics can provide in these circumstances is bad tidings. There is also a problem with adapting many complexity metrics to practical systems. McCabe and Butler (1989) note that practical systems normally have 100 times the design complexity of those used in textbooks. Low (good) complexity values are desirable, but of limited use unless a lot of ancillary information is available about the development process. All of the recommended complexity metrics (Halstead, McCabe, and Henry) will pinpoint unduly complex modules. The McCabe metric will, in addition, pinpoint unstructured code (McCabe 1976).

Alternatives available to an *a posteriori* reviewer are limited, and complexity metrics can help in making the difficult decision about which alternative to take. A preliminary assessment indicating possible difficulties may provide guidance on where to concentrate investigatory effort. High complexity and unstructured code may support a decision to reject or reverse-engineer a design. Good complexity numbers, coupled with a positive review of the organizational software development process and extensive documentation, may support a decision to proceed.

Although not recommended at this time, two metrics mentioned in this report may be good subjects for further research. These are the application of McCabe's design complexity graphs (McCabe and Butler 1989) and graph-theoretic complexity for architecture (Hall and Preiser 1984). The latter metric was specifically designed for concurrent systems. Regrettably, very little experience has been reported with either metric.

4.3 Recommendations for Future Investigation

To achieve their goals, it is recommended that NRC perform research to develop quantitative metrics for systems and software. The metrics already available for software are a start, but acceptable metric values must be established for NRC needs. The factors and attributes identified for systems and software performance provide an initial list of characteristics that can be considered in developing metrics in these areas.

References

Albrecht, A., November 1983, "Prediction: A Software Science Validation," *IEEE Trans. on Software Engineering*, Vol. SE-9, 6, pp. 639–648.

Arthur, Jay Lowell, 1985, *Measuring Programmer Productivity and Software Quality*, John Wiley & Sons.

Basili, Victor R., Selby, Richard W., Phillips, Tsai-Yun, November 1983, "Metric Analysis and Data Validation Across Fortran Projects," *IEEE Trans. on Software Engineering*, Vol. SE-9, 6, pp. 652–663.

Boehm, Barry W., 1981, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N.J.

Boehm, Barry W., Gray, Terence E., and Seewaldt, Thomas, May 1984, "Prototyping Versus Specifying: A Multiproject Experiment," *IEEE Trans. on Software Engineering*, Vol. SE-10, 3, pp. 290–302.

Brinch Hansen, P., June 1975, "The Programming Language Concurrent Pascal," *IEEE Trans. on Software Engineering*, Vol. SE-1, 2, 199–207.

British Ministry of Defence, April 1991a, "The Procurement of Safety Critical Software in Defence Equipment Part 1: Guidance," Interim Defence Standard 00-55.

British Ministry of Defence, April 1991b, "The Procurement of Safety Critical Software in Defence Equipment Part 2: Requirements," Interim Defence Standard 00-55.

Brooks, Frederick P., Jr., 1975, *The Mythical Man-Month*, Addison-Wesley Publishing Company.

Burns, Alan, and Wellings, 1990, Andy, *Real-Time Systems and Their Programming Languages*, Addison-Wesley Publishing Company.

Fidge, Colin, August 1991, "Logical Time in Distributed Computing Systems," *Computer*, pp. 28–33.

Gorla, Narasimhaiah, Benander, Alan C., and Benander, Barbara A., February 1990, "Debugging Effort Estimation Using Software Metrics," *IEEE Trans. on Software Engineering*, Vol. SE-16, 2, pp. 223–231.

Hall, N.R., and Preiser, S., January 1984, "Combined Network Complexity Measures," *IBM J. Res. Develop. 28*, 1, pp. 15–27.

Halstead, M.H., 1977, *Elements of Software Science*, Elsevier North-Holland, New York.

Henley, Ernest J., Kumamoto, Hiromitsu, 1981, *Reliability Engineering and Risk Assessment*, Prentice-Hall, Englewood Cliffs, N.J.

Henry, Sallie, and Kafura, Dennis, September 1981, "Software Structure Metrics Based on Information Flow," *IEEE Trans. on Software Engineering*, Vol. SE-7, 5, pp. 510–518.

Henry, Sallie, and Selig, Calvin, March 1990, "Predicting Source-Code Complexity at the Design Stage," *IEEE Software*, pp. 36–44.

Hoare, C.A.R., October 1974, "Monitors: An Operating System Structuring Concept," *Comm. ACM 17*, 10, pp. 549–557.

Hoare, C.A.R., August 1978, "Communicating Sequential Processes," *Comm. ACM 21*, 8, pp. 666–677.

Holt, R.C., 1983, *Concurrent Euclid, the Unix System, and Tunis*, Addison-Wesley Publishing Company.

Holt, R.C., Graham, G. S., Lazowska, E. D., Scott, M. A., 1988, *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley Publishing Company.

IEEE, 1988a, "IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE Standard 982.1-1988.

IEEE, 1988b, "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE Standard 982.2-1988.

Kleinrock, Leonard, and Tobagi, Fouad A., December 1975, "Packet Switching in Radio Channels: Part 1—Carrier Sense Multiple-Access Modes and Their Throughput-Delay Characteristics," *IEEE Trans. on Communications,* Vol. Com-23, 12, pp. 1400–1416.

Kleinrock, Leonard, November 1978, "Principles and Lessons in Packet Communications," *Proc. of the IEEE,* Vol. 66, 11, pp. 1320–1329.

Lehman, M.M., and Belady, L.A., Editors, 1985, "The Characteristics of Large Systems," *Program Evolution—The Process of Software Change, APIC Studies in Data Processing No. 27,* pp. 289–329.

Leveson, Nancy G., Cha, Steven S., Shimeall, Timothy J., July 1991, "Safety Verification of Ada Programs Using Software Fault Trees," *IEEE Software,* pp 48–59.

Lombardo, Thomas G., and Mokhoff, Nicolas, August 1981, "Shuttle Firsts Put to the Test," *IEEE Spectrum,* pp. 34–39.

McCabe, Thomas J., December 1976, "A Complexity Measure," *IEEE Trans. on Software Engineering,* Vol. SE-2, 4 , pp. 308–320.

McCabe, Thomas J., and Butler, Charles W., December 1989, "Design Complexity Measurement and Testing," *Comm. of ACM 32,* 12, pp. 1415–1425.

Metcalfe, Robert M., and Boggs, David R., July 1976, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Comm. ACM 19,* 7, pp. 395–404.

MoD, see British Ministry of Defence.

Muppala, Jogesh K., Woolet, Steven P., and Trivedi, Kishor S., May 1991, "Real-Time-Systems Performance in the Presence of Failures," *Computer,* pp. 37–47.

Nelson, Bruce Jay, May 1981, *Remote Procedure Call,* doctoral dissertation, Carnegie-Mellon University, CMU-CS-81-119.

Parnas, David L., March 1979, "Designing Software for Ease of Extension and Contraction," *IEEE Trans. on Software Engineering,* Vol. SE-5, No. 2, pp. 128–138.

Parnas, David L., December 1985, "Software Aspects of Strategic Defense Systems," *Comm. ACM 28,* 12, pp. 1326–1335.

Preckshot, George G., and Butner, David N., 1987, "A Simple and Efficient Interprocess Communication System for Actually Using a Laboratory Computer Network," *IEEE Trans. Nuclear Science N-34,* 858.

Riezenman, Michael, September 1991, "Revising the Script After Patriot," *IEEE Spectrum,* pp. 49–52.

Rombach, H. Dieter, March 1990, "Design Measurement: Some Lessons Learned," *IEEE Software,* pp. 17–25.

Sahal, Devendra, June, 1976, "System Complexity: Its Conception and Measurement in the Design of Engineering Systems," *IEEE Trans. on Systems, Man, and Cybernetics,* pp. 440–445.

Savitzky, Steven R., 1985, *Real-Time Microprocessor Systems,* Van Nostrand Reinhold Company, New York, New York.

Sha, Lui, and Goodenough, John B., April 1990, "Real-Time Scheduling Theory and Ada," *Computer,* pp. 53–62.

Sha, Lui, Rajkumar, Ragunathan, and Lehoczky, John P., September 1990, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. on Computers,* Vol. 39, 9, pp. 1175–1185.

Shoch, John F., and Hupp, Jon A., December 1980, "Measured Performance of an Ethernet Local Network," *Comm. ACM 23,* 12, pp. 711 -721.

Smith, Connie U., 1990, *Performance Engineering of Software Systems,* Addison-Wesley Publishing Company.

Wietzman, Cay, 1980, *Distributed Micro/Minicomputer Systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

Wyman, R.H., et al, December 5–9, 1983, "An Overview of the Data Acquisition and Control System for Plasma Diagnostics on MFTF-B," *Proc. of 10th Symposium on Fusion Engineering*, Philadelphia, PA.

Zhao, Wei, Ramamritham, Krithivasan, and Stankovic, John A., May 1987, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Trans. on Software Engineering*, Vol. SE-13, 5, pp. 564–577.

Glossary

Asynchronous event. An event which is not synchronized with the execution of an instruction in a computer. Typically, something external to the computer that causes an interrupt.

Branch. *IEEE Std 610.12-1990:*

(1) A computer program construct in which one of two or more alternative sets of program statements is selected for execution.

(2) A point in a computer program at which one of two or more alternative sets of program statements is selected for execution.

(3) Any of the alternative sets of program statements in (1).

(4) To perform the selection in (1).

Additional explanation:

In computer machine code, for many computer designs, the default location for finding the next instruction to execute is the next sequential location in memory. An instruction to take the next instruction from a different location, including subroutine jumps and returns, is called a branch instruction. Branch instructions may be conditional upon register values or the results of previous instructions (e.g., arithmetic overflow, parity, sign, etc.). Branches complicate code. Code which has few branches is easier to verify. It also makes fewer decisions.

Branch instructions are generated by compilers in response to conditional statements in the high-level language being compiled. For example, "if," "while," and "do" statements will result in branches.

Completion routine. A completion routine is a subprogram within a program that is executed as a result of an interrupt, but after the interrupt service routine has finished. The completion routine executes within the context of the parent program, having access to parent program data, but with separate program counter and working register values. In UNIX systems, a similar but not identical construct is called a signal handler. Both completion routines and signal handlers are forms of light-weight tasks.

Completion routines preempt parent program code, and so may access the same data items being accessed by parent code. This makes completion routines potentially critical regions, and they are often used to implement synchronization methods by passing data through interrupt-impervious data structures like ring buffers.

Concurrency. A condition where two or more instruction streams execute simultaneously or apparently simultaneously. In single processor systems, apparent concurrency occurs because a lower-priority task may be interrupted by higher-priority tasks, giving the appearance of simultaneous execution. Concurrently executing programs may attempt to access the same data, giving rise to inconsistent data values. Such regions of code (where this is possible) are called critical sections, and considerable effort must be taken to synchronize access to such code. See Hoare 1974 for further details. See Holt et al 1978 or Holt 1983 for texts on concurrent programming.

Context. Context is the entire state of an execution unit, including register contents, floating point register contents, memory mapping state, hardware priority state, and current program counter value. The number of things included in context can vary considerably depending upon Central Processing Unit (CPU) design. Some Reduced Instruction Set Computer (RISC) CPUs, for instance, may save as many as 80 32-bit quantities as context.

Context is important because it is saved and restored during interrupts and when switching between tasks. Sometimes only a limited context save is performed during interrupts. The effort and delay associated with a task switch is justified when tasking is used to modularize and separate the work of one task from another. See Savitzky 1985 for more detail on context switching.

Continuous-time simulation. A method of simulation in which dynamic systems are modeled by differential equations which are solved by various numerical analysis techniques. Time is considered a continuous parameter, and solutions are considered to be analytically continuous between solution points. This differs from discrete event simulation in that discrete event systems are defined only at event times, and represent not dynamic systems but logical systems.

Critical section. "A sequence of statements that must appear to be executed indivisibly is called a critical section. The synchronization required to protect a critical section is known as mutual exclusion" (Burns & Wellings 1990).

Data structure. *IEEE Std 610.12-1990:*

A physical or logical relationship among data elements, designed to support specific data manipulation functions.

Additional explanation:

A usually contiguous section of memory which contains some combination of elementary data types (integers, pointers, floating point numbers) and other data structures which is treated as a single variable for purposes of data modularity and argument passage. A grouping of smaller data items.

Deadlock. *IEEE Std 610.12-1990:*

A situation in which computer processing is suspended because two or more devices or processes are each awaiting resources assigned to the others.

Discrete-event simulation. A type of simulation in which the system under simulation is driven by events which occur at discrete moments in time. The system's response may generate more events which occur at future times, which the simulator captures, queues, and feeds back into the system under test. Discrete event simulators are the underlying form of simulator used for many performance model simulations, petri net simulations, and queuing network simulations.

Embedded system. *IEEE Std 610.12-1990:*

A computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system.

Additional explanation:

There is also the implication that the computer system is not programmable by its usual users and that it is not easily removable or convertible to another purpose.

Event-based. A system organizing concept (from the programmer's point of view) wherein an application program is notified of outside occurrences by events. Contrast with message-based.

Exception. *IEEE Std 610.12-1990:*

An event that causes suspension of normal program execution. Types include addressing exception, data exception, operation exception, overflow exception, protection exception, underflow exception.

Additional explanation:

Interrupts are considered a type of exception, but not the reverse. An exception not only suspends normal program execution, but saves at least the location in program where the exception occurred (so as to return to it later) and restarts the processor at the beginning of exception handling code. In

some designs, exception processing by hardware is quite complex, and a lot of processor context is saved automatically by hardware. In other designs, notably most RISC designs, only the bare minimum of context is saved.

In some languages, Ada for example, exceptions are a language construct which allow software to add exceptions to those already produced by hardware. See synchronous event.

Foreground/background. A simplified form of prioritized multi-tasking consisting of two tasks, the foreground task and the background task. When the foreground task is ready to run it preempts the background task. This is one of the more common arrangements for simple real-time systems.

Function. *IEEE Std 610.12-1990:*

(1) A defined objective or characteristic action of a system or component. For example, a system may have inventory control as its primary function.

(2) A software module that performs a specific action, is invoked by the appearance of its name in an expression, may receive input values, and returns a single value.

Interprocess communication. Communication between processes or tasks which are executing asynchronously with respect to each other, in different contexts, and possibly on different processors. In any interprocess communication there is inherent synchronization in that the sender must send before the receiver receives. Progressively more strict synchronization disciplines may be imposed, e.g., broadcast, one-to-one, rendezvous, remote procedure call.

In light of the definition of process, interprocess communication is probably a misnomer, and should be styled intertask communication.

Interrupt. *IEEE Std 610.12-1990:*

(1) The suspension of a process to handle an event external to the process.

(2) To cause the suspension of a process.

(3) Loosely, an interrupt request.

Additional explanation:

Interrupts are considered exceptions caused by asynchronous events, whereas traps and other exceptions are caused by synchronous events. Like other exceptions, hardware usually saves some portion of context, minimally a return address.

Inter-task communication. Synonymous with interprocess communication.

Kernel. *IEEE Std 610.12-1990:*

(1) That portion of an operating system that is kept in main memory at all times.

(2) A software module that encapsulates an elementary function or functions of a system.

Additional explanation:

A misnomer when applied to the UNIX operating system.

Light-weight task. A thread which can execute in parallel to the main thread of a parent task, while sharing some of the context of the parent task. It is called a light-weight task because only a limited context switch is required to switch between it and its parent.

Message-based. A system organizing concept (from the programmer's point of view) wherein an application program is notified of outside occurrences by the reception of messages. Contrast with event-based.

Multi-tasking. *IEEE Std 610.12-1990:*

A mode of operation in which two or more tasks are executed in an interleaved manner.

Additional explanation:

A term usually applied to an operating system which can execute two or more tasks apparently concurrently (i.e., interleaved). The advantage of this is that each task can follow an external physical process independently of other tasks following other physical processes that have significantly different time histories.

Multi-threaded. Having two or more code threads which can be executed apparently concurrently. Usually applied to code residing within a task.

Petri net. *IEEE Std 610.12-1990:*

An abstract, formal model of information flow, showing static and dynamic properties of a system. A Petri net is usually represented as a graph having two types of nodes (called places and transitions) connected by arcs, and markings (called tokens) indicating dynamic properties.

Polling loop. A program organization technique by which a list of inputs is traversed repeatedly and tested for readiness. Each ready input is dealt with on the spot and then the program proceeds to the next input in list order. This has the advantage of extreme simplicity.

Preemption. Suspending a task (usually because of an interrupt), saving its context, and starting a higher priority task in its place.

Priority. *IEEE Std 610.12-1990:*

The level of importance assigned to an item.

Additional explanation:

An integer ordering placed on tasks which governs a strict rule of preemption. Priority can be static or dynamic. Static priority is fixed at the creation of a task. A task with dynamic priority can have its priority changed by fairness schemes during execution or by inheritance (e.g., it is blocking a higher-priority task due to resource contention.)

Procedure. *IEEE Std 610.12-1990:*

(1) A course of action to be taken to perform a given task.

(2) A written description of a course of action as in (1); for example, a documented test procedure.

(3) A portion of a computer program that is named and that performs a specific action.

Additional explanation:

A procedure is a program or subprogram which performs a sequence of computations or input/output operations. As a subprogram, a procedure is invoked with arguments (and possibly global variable values) and returns results in the same or other arguments (and possibly in global variables). In some high-level languages, a distinction is made between procedures and functions, with functions returning a value and procedures not.

Process. *IEEE Std 610.12-1990:*

(1) A sequence of steps performed for a given purpose; for example, the software development process.

(2) An executable unit managed by an operating system scheduler.

(3) To perform operations on data.

Additional explanation:

Process is sometimes used interchangeably with task, depending upon whose documentation is being read. The IEEE standard (1) defines the word with the majority; process is what is done, task is a computer-mechanized instance of doing it.

Protocol. *IEEE Std 610.12-1990:*

A set of conventions that governs the interaction of processes, devices, and other components within a system.

Additional explanation:

A set of standard formats for encapsulating data sent through a network so that the data can be routed, sequenced, reassembled, transformed, and delivered reliably and correctly. Communication protocols mediate between systems, not within systems.

Real-time system. The IEEE Std 610.12-1990 definition omits the mention of deadlines, which are central to current real-time theory.

Real-time systems are digital systems which must produce correct output in response to input within well-defined deadlines (Burns & Wellings 1990).

(1) Real-time systems have inputs connected to real-world events.

(2) Events (inputs) cause computations to occur which finish before well-defined deadlines, measured from the event.

(3) Computations produce outputs that are connected to the real world.

Software for real-time systems has an additional correctness constraint: even if the values produced by software are correct, the software is considered to fail if it does not meet its deadline. Current-day practice further divides real-time systems into hard real-time systems and soft real-time systems. Hard real-time systems cannot tolerate any failures to meet deadlines. Soft real-time systems can tolerate some deadline failures and will still function correctly.

Remote procedure call. A restricted rendezvous with the semantics of procedure call (Nelson 1981). The caller does not depart until it has an answer from the callee.

Rendezvous. A synchronization discipline for sending messages between tasks. Both the sender and the receiver arrive (and wait) at a rendezvous until they can simultaneously depart. The sender departs knowing that the receiver received and the receiver departs knowing this also. This has the advantage of providing absolute knowledge to the involved tasks that the rendezvous was successful, but may leave either waiting forever.

Resource. Generally, a resource is any identifiable object or capability of a computer system required to get a job done and that can potentially be shared among two or more jobs. Objects can include devices, portions of memory, or shared variables. Capabilities can include CPU time or datalink capacity. The general meaning is the one applied when adding up or estimating needed resources for performance estimation (Smith 1990).

Formally in a specific computer system implementation, a resource is a named object which is allocated to one or more processes requiring it and released as processes are finished with it (Burns and Wellings 1990). For example, a printer may be allocated to a job printing a file and released at the end of the file. The allocation and release protocol prevents portions of other files from appearing intermixed in the file currently being printed.

Run-time kernel. A kernel generally provided for a real-time embedded system. The implication is that it is fast, has many good real-time features, and provides a basis upon which to build a real-time application.

Scalability. The property that describes the ease of extension of small to large. Scalability includes both the extension of actual small systems to large systems and the application of small system techniques inappropriately to the large system domain.

Shared memory. Physical memory that is attached to two or more processors by multiple ports on the memory module or through a shared bus with hardware contention arbitration. Speed of access is the same or slightly slower than for memory attached exclusively to one processor.

Signal. A UNIX abstraction which sometimes acts like a trap, an exception, or an interrupt.

Software structure chart. *IEEE Std 610.12-1990:*

(1) A diagram that identifies modules, activities, or other entities in a system or computer program and shows how larger or more general entities break down into smaller, more specific entities. Note: The result is not necessarily the same as that shown in a call graph.

(2) Call graph—a diagram that identifies the modules in a system or computer program and shows which modules call one another.

Additional explanation:

For many people, structure chart means call graph.

Synchronization. The term "synchronization" is overloaded in electrical engineering and computer science, which may reflect its importance. To place the term in perspective, its meanings in a variety of contexts are given:

Communications: maintaining phase lock between a multiple of the frequency of a carrier or subcarrier and demodulation circuitry in a receiver.

Computer science/operating systems: queuing an event or data item so that it can be handled later by software that is already busy doing something. When the handler software is ready to service a new request, it asks for the next event or data.

Computer science/distributed systems: maintaining time or event order at physically separated points in a distributed system.

Synchronous event. An event that occurs as a result of executing a low-level (machine language) computer instruction. Typically, this might be an exception due to an instruction fault, memory mapping error, or deliberate system call.

Task. *IEEE Std 610.12-1990:*

(1) A sequence of instructions treated as a basic unit of work by the supervisory program of an operating system.

(2) In software design, a software component that can operate in parallel with other software components.

Additional explanation:

Task also conveys the implication of independence. Tasks usually execute in their own context.

Thread. A sequence of instructions which has no parallel components. The current position in the thread can be represented by a single number (address). Threads are currently a part of the IEEE P1003.4a Posix draft standard for real-time enhancements to portable operating systems.

Trap. *IEEE Std 610.12-1990:*

(1) A conditional jump to an exception or interrupt handling routine, often automatically activated by hardware, with the location from which the jump occurred recorded.

(2) To perform the operation in (1).

Additional explanation:

Trap comes from the expression "trapped instruction," which early computer makers used to implement in software those instructions they had been unable to commit to hardware. Trap is now sometimes used interchangeably with interrupt and exception.

NRC FORM 335 (2-89) NRCM 1102, 3201, 3202

U.S. NUCLEAR REGULATORY COMMISSION

BIBLIOGRAPHIC DATA SHEET

(See instructions on the reverse)

1. REPORT NUMBER (Assigned by NRC, Add Vol., Supp., Rev., and Addendum Numbers, if any.)
NUREG/CR-6083
UCRL-ID-114565

2. TITLE AND SUBTITLE
Reviewing Real-Time Performance of Nuclear Reactor Safety Systems

3. DATE REPORT PUBLISHED
MONTH: August
YEAR: 1993

4. FIN OR GRANT NUMBER
L1867

5. AUTHOR(S)
G. G. Preckshot

6. TYPE OF REPORT
Technical

7. PERIOD COVERED (Inclusive Dates)

8. PERFORMING ORGANIZATION – NAME AND ADDRESS (If NRC, provide Division, Office or Region, U.S. Nuclear Regulatory Commission, and mailing address; if contractor, provide name and mailing address.)
Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, CA 94550

9. SPONSORING ORGANIZATION – NAME AND ADDRESS (If NRC, type "Same as above"; if contractor, provide NRC Division, Office or Region, U.S. Nuclear Regulatory Commission, and mailing address.)
Division of Reactor Controls and Human Factors
Office of Nuclear Reactor Regulation
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001

10. SUPPLEMENTARY NOTES

11. ABSTRACT (200 words or less)

The purpose of this paper is to recommend regulatory guidance for reviewers examining real-time performance of computer-based safety systems used in nuclear power plants. Three areas of guidance are covered in this report. The first area covers how to determine if, when, and what prototypes should be required of developers to make a convincing demonstration that specific problems have been solved or that performance goals have been met. The second area has recommendations for timing analyses that will prove that the real-time system will meet its safety-imposed deadlines. The third area has descriptions of means for assessing expected or actual real-time performance before, during, and after development is completed. To ensure that the delivered real-time software product meets performance goals, the paper recommends certain types of code-execution and communications scheduling. Technical background is provided in the appendix on methods of timing analysis, scheduling real-time computations, prototyping, real-time software development approaches, modeling and measurement, and real-time operating systems.

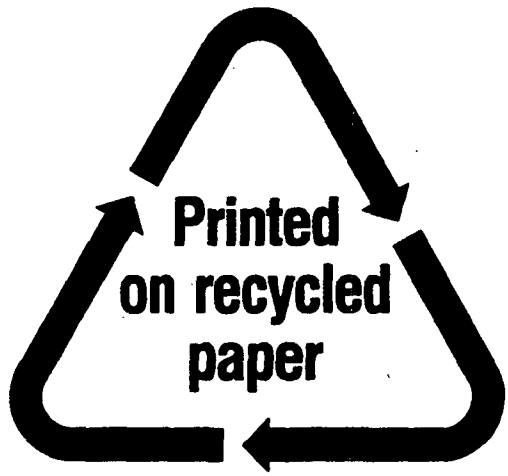
12. KEY WORDS/DESCRIPTORS (List words or phrases that will assist researchers in locating the report.)
Real-Time Performance
Computer-Based Safety Systems
Prototype Testing
Complexity (System and Software)

13. AVAILABILITY STATEMENT
Unlimited

14. SECURITY CLASSIFICATION
(This Page)
Unclassified
(This Report)
Unclassified

15. NUMBER OF PAGES

16. PRICE



Federal Recycling Program

UNITED STATES
NUCLEAR REGULATORY COMMISSION
WASHINGTON, D.C. 20555-0001

OFFICIAL BUSINESS
PENALTY FOR PRIVATE USE, $300

SPECIAL FOURTH-CLASS RATE
POSTAGE AND FEES PAID
USNRC
PERMIT NO. G-67