Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives

Connie U. Smith, *Senior Member, IEEE*, and Lloyd G. Williams

Abstract— **Software Performance Engineering (SPE) provides an approach to constructing systems to meet performance objectives. This paper illustrates the application of SPE to an example with some real-time properties and demonstrates how to compare performance characteristics of design alternatives. We show how SPE can be integrated with design methods and demonstrate that performance requirements can be achieved without sacrificing other desirable design qualities such as understandability, maintainability, and reusability.**

Index Terms— **Ada, Ada performance, design, domain analysis, object-oriented, software performance engineering, software performance models.**

I. Introduction

PERFORMANCE is an important but often neglected aspect of software development methodologies. Performance refers to system responsiveness: either the time required to respond to specific events, or the number of events processed in a given time interval. For traditional information systems, performance considerations are often associated with useability issues such as response time for user transactions. For "soft" real-time systems, performance considerations relate to the ability to adequately handle the external load placed on the system. In "hard" real-time systems, performance concerns can become correctness issues. In these systems, failure to meet deadlines or throughput requirements is equivalent to producing incorrect results.

Traditional software development methods have taken a "fix-it-later" approach to performance. "Fix-it-later" advocates concentrating on software correctness; performance considerations are deferred until the later phases of the software process (i.e., integration and testing). If performance problems are discovered at this point, the software is "tuned" to correct them and/or additional hardware is used.

The "fix-it-later" approach is clearly undesirable. Performance problems may, for example, be so severe that they require extensive changes to the system architecture. If these changes are made late in the development process, they can increase development costs, delay deployment, or adversely affect other desirable qualities of a design, such as understandability, maintainability, or reusability. Finally, designing for performance from the beginning produces systems that exhibit better performance than can be achieved using a "fix-it-later" approach.

In this paper we describe and illustrate the application of Software Performance Engineering (SPE) techniques to the management of performance concerns in the development of software-based systems. SPE provides guidelines for applying well-established performance modeling techniques throughout the development process to help ensure that the resulting software system meets its performance requirements. SPE methods cover performance data collection, quantitative analysis techniques, prediction strategies, management of uncertainties, data presentation and tracking, model validation and verification, critical success factors, and performance design principles [53]. SPE originated in the capacity planning and performance evaluation communities and has been applied primarily to data processing systems. Here, we illustrate the application of SPE techniques to an example with some real-time properties. This paper has three principal goals: to illustrate the application of SPE in the software development process; to show how SPE can be integrated with design methods; and to demonstrate that performance requirements can be achieved without sacrificing other desirable design qualities. We also illustrate extensions to performance modeling techniques to represent certain features of the Ada language.

The concepts of performance engineering are developed and illustrated via a simple case study. To keep the presentation to a manageable size, we discuss the application of SPE during the design phase. Section II of this paper discusses related work. Section III provides some background on the design approach used for the case study and SPE techniques. Section IV presents the case study and the design and Section V describes its performance analysis. Section VI discusses the results and compares them with results for other design alternatives for this case study.

II. Related Work

A number of other projects have addressed techniques that can be used for evaluating and comparing the performance characteristics of software designs. In this section, we classify and review those that are most relevant to our work, and conclude by contrasting the approaches.

Manuscript received May 7, 1992; revised January 19, 1993. Recommended by Stuart Zweben.

C. U. Smith is with Performance Engineering Services, Santa Fe, NM 87504.

L. G. Williams is with Software Engineering Research, Boulder, CO 80302.

IEEE Log Number 9210754.

- *Performance modeling tool approaches:* This category covers the use of new or existing tools to model and analyze the performance of software systems. Some use queueing network models to represent contention and synchronization and usually solve them with simulation methods. Representative examples include [58], [34], [40], and [3]. Similar examples also appear in the winter and summer simulation conferences. Other approaches use Petri net models for performance analysis [18], [15]. These are discussed separately below.
- *Petri net approaches:* This category includes work that uses Petri nets to represent designs and analyze their performance. Some work provides tools to enable timed or stochastic analysis, such as [18], [15], [14]. Some extend the Petri nets to also represent safety, reliability, or performability, such as [35], [38], and [56]. Other work, such as [7], uses Petri nets as a design notation, thus enabling performance analysis. Some work, such as [6], integrates the Petri net design notation into a CASE tool that provides performance results. There is extensive work in the Petri net category.
- *Quantitative models:* This category includes work that provides fundamental performance modeling results that enable the application of SPE techniques to software systems. Many researchers contributed to the early modeling techniques that established the viability of SPE. Their work is reviewed in [55]. Ammar and Qin present new software model solution algorithms in [1]. Sholl and Kim have applied their computation structure techniques to real-time systems [52]. Joseph and Pandya claim a computationally efficient technique for finding the exact worst-case response time of real-time systems [29].

  Rolia extends performance models and methods to address systems of cooperating processes in a distributed and multicomputer environment with specific application to Ada [46]. Woodside and co-workers propose stochastic rendezvous nets to evaluate the performance of Ada systems and incorporate them into the CAEDE environment [63], [62], [10]. Kant develops Extended Petri Nets that combine queueing transitions and synchronization transitions to quantify software synchronization and contention delays [30]. He develops special solution algorithms for a variety of interaction scenarios.
- *CASE tools:* Computer-aided software (or systems) engineering (CASE) provides tools for constructing models, at various levels of detail, of systems under development. Model analysis can include: syntax checking, completeness and consistency checking, and data model normalization. Dynamic analysis can include reachability, deadlock, and nontermination. Some CASE tools provide performance analysis capabilities. For example, CardTools analyzes performance in the latter stages of design or implementation for systems implemented in Ada in their target environment (VRTX/ARTX) [45]. The Teamwork CASE tool interfaces with the ADAS performance simulator and provides visualization of performance results [12].
- *Formal methods:* Formal methods are approaches, based on the use of mathematical techniques and notations, for describing and analyzing properties of software systems. Typically, descriptions or models of the software are constructed using a formal mathematical notation rather than natural language. The descriptions are then analyzed, perhaps using inference techniques or proofs of correctness, for completeness, consistency and other properties. Formal methods exist that can describe some aspects of real-time systems, such as concurrency. Other issues, such as timing, are still open research topics [16]. Proposals for including time in formal descriptions of real-time systems include [26], [27], and [20]. These approaches provide assistance in specifying timing properties of real-time systems but do not provide guidance in how to meet deadline requirements.
- *System-oriented approaches:* This category focuses on operating system executives geared to real-time systems. Some work also provides guidance to designers for the creation of application software to operate effectively with the executive. Kopetz and co-workers take this approach in their MARS system [31]; [32]; Levi and Agrawala propose their MARUTI system [36]; Baker and Scallon propose REX [4]; and Stankovic and Ramamritham advocate their Spring Kernel [57]. To capitalize on this work, developers must use the entire system.
- *Resource allocation approaches:* Resource allocation approaches attempt to guarantee that a certain subset of a system's tasks always meet their execution deadlines. Approaches in this category focus on scheduling the various processes so that critical processes will always meet their timing constraints. The scheduling may be assigned statically or computed dynamically. Xu and Parnas [64] review several of these techniques. Dynamic scheduling approaches must rely on imperfect knowledge of the execution characteristics of arriving processes. These techniques, therefore, sacrifice CPU utilization to guarantee schedulability (see, e.g., [50]).

Software Performance Engineering provides techniques that unify and augment these diverse approaches. Many authors document SPE methods, models, and experience for data processing systems (see [53] and references therein). Fox describes practical experience in applying SPE methods to systems during development [19]. He emphasizes the use of measurements and performance testing throughout implementation. While Fox does not specifically address systems with real-time constraints, his advice does apply to them.

In contrast to system-oriented approaches, SPE focuses on applications running on any executive and uses quantitative techniques to manage the overall performance of systems and applications. Static resource allocation approaches must rely on a number of assumptions to make the mathematical problems tractable. Performance modeling can help establish the validity of those assumptions. SPE prescribes a systematic approach for integrating performance assessment with design. It is not limited to specific problem domains or to any particular performance modeling tool; any of the quantitative models mentioned above would be appropriate. SPE uses the

most appropriate performance modeling techniques at each life cycle stage.

While the capabilities offered by current-generation CASE products help to advance the ability to apply SPE during software design, they still require considerable supplemental work to create and analyze performance models. The specific performance modeling steps (when to assess performance, and what should be assessed) are not addressed by the CASE methods and tools [54].

The use of Petri nets as a design notation is a matter of personal taste. Petri nets excel at the representation and behavioral analysis of synchronization and communication among concurrent tasks. However, they require extensions to represent contention delays at shared computer resources (see, e.g., [37], [5], [22]). Furthermore, the time required for solution of complex Petri net models makes them inconvenient for design tradeoff studies. The SPE philosophy suggests using simpler models early in development to identify suitable designs. As the software evolves, model extensions analyze contention delays, synchronization and communication behavior. During implementation, the SPE models (described in this paper) can produce Petri net models that permit extensive, detailed analysis of timing, synchronization and communication. Some designers prefer a more abstract initial representation while others prefer the detail and precision of Petri nets. This paper presents the SPE approach which advocates the use of detailed models during the later stages of development. We focus on the early life cycle analyses, and the Petri net approaches are, consequently, beyond the scope of this paper.

Formal methods can complement SPE by providing a means of unambiguously specifying timing requirements. SPE then provides techniques for assessing the feasibility of implementing these requirements. SPE also provides coverage throughout the development process to help ensure that the requirements are achieved.

SPE uses simple stochastic models early in development to identify and correct problems early. Some researchers oppose the use of stochastic modeling approaches. Xu and Parnas draw an analogy between modeling and testing, both of which only detect the presence of problems but do not prove their absence [64]. SPE advocates using quick, easy techniques to detect problems first then, later, proceeding to the other analyses.

III. Background

This section presents background information on the approach used to produce the case study design and Software Performance Engineering techniques used to analyze it. Our intent is to provide enough information so that the reader can appreciate the interplay between design considerations and SPE techniques.

A. Design Methodology

The systems analysis and design approach used here relies on domain analysis [44], [2] and object-oriented development. The goals of the development effort were to construct a design that is easy to understand and modify, has a high potential for reuse, and provides adequate performance. The domain analysis process is described in detail in [60]. Because we wish to focus on the performance characteristics of the design, this section provides only an overview of the technique.

The domain analysis technique is based on the identification and modeling of multiple domains [59]. This approach is motivated by the observation that any completed application actually includes components from several different problem domains. The most obvious domain is the central problem area, or *dominant domain*, such as industrial process control or avionics. Other domains, known as *supporting domains*, while not strictly part of the dominant domain are required for an implementation. Examples of supporting domains include user interfaces, digital data acquisition, and communications. Supporting domains may be valid application domains in their own right. In the context of the multiple domain approach, however, their role is to support the overall purpose of the dominant domain. Identification of these different types of information, followed by partitioning into the appropriate domains can produce specifications and designs which are more understandable, easier to modify, and which contain components that have a higher potential for reuse than those produced without domain analysis.

The analysis process consists of identifying the various problem domains that contribute to the overall system. Each of these domains is then modeled using an object-oriented strategy. The technique used here is similar to that advocated by a number of other authors, including [51], [47], and [17]. The models consist of: an Information View, modeled using an extended entity-relationship notation; a Behavior Pattern View, modeled using state-transition diagrams; and a Process View, modeled using real-time extensions to data-flow diagrams [61]. Briefly, the Information View is used to identify the classes of objects in the domain and model their structural characteristics. For each class that has a nontrivial behavior pattern, a Behavior Pattern View is constructed. The Process View describes the operations (methods) that are applicable to the class. Details of the modeling strategy, as applied to the case study presented below, are presented in [60].

Individual components are designed for each domain. These components are then integrated to produce an overall system design. The specific approach to integration depends somewhat on the target environment. In an Ada environment, for example, components from supporting domains may be instantiated from generic packages. Here, integration is concerned with both interface and instantiation issues. In an object-oriented language, a supporting component may be designed as a subclass of an abstract class defined by the dominant domain.

B. SPE Basics

Software Performance Engineering relies on models to predict the performance of early software plans. The type of performance model appropriate for an SPE study depends on the purpose of the analysis and the precision of the input data. Three analysis strategies guide the model formulation:

- *Adapt-to-precision strategy:* match the modeling effort to available knowledge of system details. Early in devel-

opment, the purpose of SPE is to identify development plans that can meet performance objectives and, when multiple alternatives exist, to provide data for analysis that will help select the most desirable alternative. At this stage, designers focus on high-level decisions rather than implementation details. The adapt-to-precision strategy, therefore, suggests that analysts use models that can be constructed quickly, enable rapid evaluation of alternatives, and produce results that distinguish suitable plans from those that are unlikely to meet performance objectives.

- *Simple-to-realistic strategy:* start with simple, high-level models and add details, one at a time, as the system evolves. Early models and their results need to be easy to understand and explain. Therefore, the simple-to-realistic strategy suggests that analysts postpone modeling details of execution. Later as developers make specific design and implementation decisions, these simple models can be extended to more precisely represent execution behavior.
- *Best-and-worst-case strategy:* examine bounds on system performance. Early performance studies seldom have precise input data, so the best-and-worst-case strategy is used to identify potential risks. If the performance predictions based on best-case data do not meet objectives, developers must seek feasible alternatives before proceeding. If the worst-case data produces satisfactory performance predictions, they can proceed to the next phase of development. If the results fall between the two extremes, SPE methods prescribe a variety of techniques to identify and focus on critical components to obtain more precise data and to monitor and manage critical-component performance as the system evolves.

In this paper we show how these strategies drive SPE model creation. We illustrate models of intermediate complexity appropriate for performance prediction during software design for systems with real-time characteristics.

In order to construct and evaluate SPE models, analysts need five types of data:

- *Performance requirements:* Specific, quantitative performance requirements for the system must be defined. These requirements identify events of interest and time constraints on the handling of those events.
- *Behavior patterns and intensity:* This requires identification of the distinct types of events that occur as well as their intensity. Intensity is defined either in terms of the arrival rate of each type of event or the number of concurrent users (or external entities) that interact with the system and the frequency of their requests (events). Specifications for both steady-state and worst-case behavior are needed.
- *Software descriptions:* The operations that provide responses to the events must be described. The level of detail increases as the software evolves.
- *Execution environment:* The execution environment is described by specifying the hardware devices and (significant) software service routines required for execution and the service rates for each of them. Any other work that may introduce resource contention delays must also be included in the execution environment.
- *Resource usage estimates:* Resource requirements for operations, in terms of processor demands, I/O, and (later) memory requirements, must be quantified. These provide the estimated number of service requests per device or service routine and the amount of service needed for each operation.

Software Performance Engineering represents the data with two models, the *software execution model*, and the *system execution* model. They differ in degrees of sophistication depending on the level of knowledge about the system under development and the requirements of the task at hand. The following sections describe the models.

C. Software Execution Models

The SPE process begins with *software execution models* that capture essential aspects of the software performance behavior. Analysts construct software execution models for typical workload scenarios which specify the operations to be executed in response to predefined events.

A workload scenario also specifies the frequency with which those operations occur as well as any special protocol, synchronization, or communication with other operations. Following the simple-to-realistic strategy, these models are easily solved. They provide initial indications of whether the proposed software will meet performance goals and are used to derive parameters for the system execution model (described below).

Software execution models are constructed using *execution graphs.* Nodes in an execution graph represent software components while arcs represent transfer of control. Simple execution graph models use four types of nodes. Basic nodes represent components at the lowest level of detail appropriate for the model; expanded nodes represent components that are elaborated in a subgraph at a lower level of detail. Repetition and case nodes add iteration and selection to the sequential control provided by arcs. These execution graphs and their solution are defined in [53].

Service requirements for each node in the execution graph are determined from descriptions of the execution environment together with best-and-worst-case resource estimates. The model solution yields total average service requirements per device for each workload scenario. We start with specifications for

$r_i \quad i = 1, \cdots, k$ The requirements for each resource i for a node.

and use graph reduction rules to compute

$R_i \quad i = 1, \cdots, k$ The total units of service for each computer resource i for the scenario.

The graph reduction rules are applied iteratively to reduce the graph one step at a time. Each step identifies a path structure (sequential, conditional or repetition paths) and applies a reduction formula to compute r_i for the structure. Iterative

application of the formulas reduces the graph to a single node with the computed R_i for each resource i.

Software execution models are straightforward to construct and can be used to prepare spreadsheet models which can quickly quantify resource requirements under a variety of conditions. A typical use for these models is best-and-worst-case analysis. In these studies, optimistic and pessimistic estimates are used for the resource requirements of the nodes in a scenario. As described above, if these estimates indicate that there are no problems, analysts can proceed to more detailed and sophisticated models.

D. System Execution Models

After making any software or hardware configuration changes which are indicated by the best-and-worst-case analysis, the execution graph data is used to derive parameters for a *system execution model.* System execution models represent the key hardware devices with queue-servers and connect them with arcs to show possible paths though the system. Execution scenarios (from the execution graphs) translate into model workloads.

The parameters for system execution models are:

- *Transaction categories:* workload elements, j, with distinct execution characteristics (routing through or visits to queue-servers and service requirements per visit)
- *Workload intensity:* number of concurrent tasks per category, M_j, or arrival rates for events in each category, λ_j
- *Service requests:* number of visits per node i per category, V_{ij}
- *Service time:* average per node i per category, S_{ij}.

The model solution produces metrics that quantify the effect of resource contention delays on each workload. Sensitivity studies explore performance bounds on resource usage, ranges of workload intensities, and various combinations of competing workloads as well as check for problem areas. After correcting any problems identified by simple execution models, more advanced models examine synchronization and communication among execution scenarios. System execution models may be constructed using Information Processing Graphs (IPG's)—a graphical representation of queuing networks [41]. Details on the use of IPG's for system execution models are in [53].

Analytic algorithms for solving elementary system execution models such as this are in many sources (see [33], [49], [39], [28]). The model solution produces metrics for device residence times (service time plus expected queuing delays), device utilizations, throughputs, queue lengths, and overall response times. Analytic solution techniques produce mean-value results. If the mean values do not meet performance objectives, developers seek remedies before proceeding to more advanced system execution models.

IV. Case Study

The case study in this paper is an adaptation of a problem originally presented in [65]. The description given below is from [42]. This case study was chosen for several reasons. It is straightforward enough to be understandable in a short presentation. Despite its simplicity, however, the case study exhibits a number of features and problems typical of real-time and reactive systems.

This case study also has one additional advantage; two previous papers proposed different designs for this problem. The first, presented by Nielsen and Shumate [42] did not explicitly address performance. Performance considerations were, however, implicit in their design. The second, which was substantially different from that of Nielsen and Shumate, was presented by Sanden [48]. Sanden claimed that his design was "more efficient" based on the number of rendezvous required to make and report a temperature reading (four versus eight for the Nielsen and Shumate design). His design, however, had 15 more concurrent processes than that of Nielsen and Shumate. We thus have an opportunity to compare the performance characteristics of several alternative designs and illustrate how this information might influence the decision-making process during design.

This problem has also been addressed by Howes [23]. The object of this study, however, appears to have been to maximize the number of readings that could be obtained in a given time interval. This led to changes which were outside the scope of the original problem description. The communication protocol, for example was redefined to eliminate the need for an interrupt each time a character is transmitted. Howes' design cannot, therefore, be directly compared with the one presented in this paper or those presented by Nielsen and Shumate or Sanden.

A. Case Study Description

The Remote Temperature Sensor (RTS) measures the temperature of a number of furnaces and reports those measurements to a remote host computer. The temperature of each furnace is obtained by periodically reading a digital thermometer. The interval between temperature readings (10–99 s) for each furnace is specified by the host computer.

A single digital thermometer is used to read temperatures for all of the furnaces. It accepts a reading request in the form of a furnace number (an integer in the range 0–15). After a request has been received, the thermometer stores the temperature of that furnace (an integer in the range 0–1000° C) in a hardware buffer and generates an interrupt. The digital thermometer can only read the temperature of one furnace at a time.

The interval between readings for a given furnace is specified in a control packet which is sent from the host computer to the RTS. The temperature of a furnace is reported to the host via a data packet. The control and data packets are sequences of ASCII characters which have the following formats:

Control Packet: (STX)(FF)(SS)(ETX)
Data Packet: (STX)(FF)(TTTT)(ETX)

where

STX is the start of text indicator
FF is the furnace number
SS is the minimum number of seconds between readings
TTTT is the temperature
ETX is the end of text indicator.

When a control packet is received from the host, it is checked for validity. The message is valid if it has an STX and its two fields contain numbers which are within the proper range. If the message is valid, an acknowledgment (ACK) is sent to the host. If it is not valid, a negative acknowledgment (NAK) is sent.

When a data packet is sent, the RTS waits to receive either an ACK or a NAK from the host. If an ACK is received, a new message may be sent. If a NAK is received, the message is retransmitted. If neither an ACK nor a NAK is received within a 2-s timeout period, the message is retransmitted.

B. Problem Analysis

This section provides an overview of the problem analysis. Details, including information, behavior pattern, and process models may be found in [60].

The problem statement, as is the case with most requirements documents, mixes problem domain information with implementation details. For example, the composition of the data and control packets and their transmission protocols are information that is more properly part of an implementation than the problem domain itself.

Closer examination of the problem statement reveals that it contains information about three different problem domains, each of which contributes to the ultimate solution. The dominant domain, apparently part of an industrial process control application, is concerned with measuring the temperatures of a set of furnaces. There are also two supporting domains, digital data acquisition (for acquiring the temperature from a digital thermometer) and communications (for interaction with the remote host).

While the RTS is likely to be part of a larger industrial process control application, the problem description indicates only two classes of objects from the dominant domain: FURNACE and THERMOMETER. FURNACE objects have a simple, but significant, behavior pattern. Initially, a FURNACE is idle and no readings are made. When a control packet is first received, a reading is made and reported. After that, a reading is made and reported at the end of each specified interval or whenever a new control packet is received. Objects belonging to the class THERMOMETER do not have significant behavior patterns. They do, however, have one relevant operation, READ, which is used by FURNACE objects to obtain their temperature readings.

The digital thermometer is a device which converts an analog temperature sensor reading to a digital value. Typically, a single analog-to-digital converter (ADC) is multiplexed to a number of sensors. Writing the sensor (or, in this case, FURNACE) number to the ADC causes it to switch channels to read the appropriate sensor. When the reading is available, an interrupt is generated. The reading is then converted to a temperature value using the sensor calibration.

The capabilities described for the digital thermometer are not limited to measurement of temperatures; many other quantities (e.g., pressure) could also be measured by changing sensors and providing an appropriate calibration. Recognition of this fact leads us to model a reusable, generic component from the domain of data acquisition. There are three classes of objects in this model: ADC, SENSOR, and SENSOR_SPEC. Only the ADC has an interesting behavior pattern. A SENSOR_ID is provided to the ADC which then selects the required sensor, makes a READING and places it in a buffer. The ADC then generates an interrupt (ADC_INTERRUPT) to indicate that the READING is available. The class SENSOR_SPEC has a single operation, GET_SENSOR_VALUE, which is used to convert the ADC READING to the appropriate value (e.g., temperature, pressure, etc.). Objects from the class SENSOR are realized completely in hardware and are, therefore, not included in the model.

This model describes a generic data capture utility since supplying new physical sensors and their corresponding calibrations (in a new SENSOR_SPEC) changes only the nature of the measurements made and the values obtained. The domain model for data acquisition may, therefore, be reused for many different applications simply by providing the proper SENSOR_SPEC.

In order to provide for interaction between the RTS and the host computer we need to introduce an additional component, the RTS_HOST_INTERFACE. This component is also a specific instance of a component from a more general domain-communications. By abstracting the control and data packets described above as input and output packets, it is possible to define a generic communications utility which is also potentially reusable. There are four classes of interest in this utility: INPUT_MESSAGE, OUTPUT_MESSAGE, INPUT_PORT, and OUTPUT_PORT.

The behavior patterns for the INPUT_MESSAGE and OUTPUT_MESSAGE classes are determined by the communications protocols for control and data packets given in the problem description. Objects belonging to the class INPUT_MESSAGE are idle until a new message is received from the INPUT_PORT. The message is then checked to determine if it is valid. If it is valid, an ACK is sent; if it is not, a NAK is sent. In either case, the object then returns to the idle state until the next message is received. The behavior pattern for OUTPUT_MESSAGE is similar. An OUTPUT_MESSAGE object is idle until a message is ready to be sent. When a new message is available, it is sent to the OUTPUT_PORT. If an ACK is received within the 2-s timeout period, the object returns to the idle state. If a NAK is received or the timeout expires (i.e., neither an ACK nor a NAK is not received within the 2-s timeout period), the message is retransmitted.

While the problem description does not provide information on the nature of the input and output ports, it is reasonable to assume that message packets are transmitted in ASCII format, one character at a time.[1] To receive a message packet, the INPUT_PORT must accept individual characters until a complete packet (a MESSAGE_PACKET, an ACK, or a NAK) has been received and then route the packet to the appropriate receiver. The arrival of a character is indicated by an interrupt generated by the underlying communications hardware. To

[1] This is the assumption made by Sanden. While other interpretations are possible (see, e.g., [23]) we follow suit to ensure that our design and Sanden's are comparable.

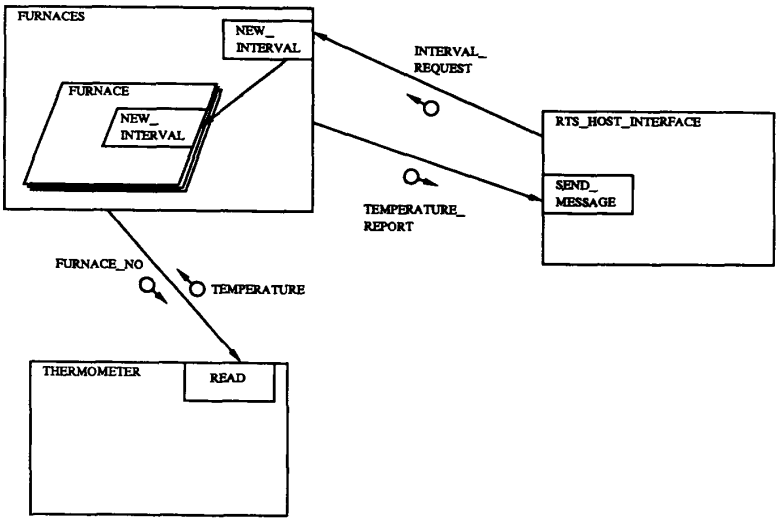


Fig. 1. Design for dominant domain.

send a message, the OUTPUT_PORT must send individual characters until the complete packet has been transmitted. A character is sent by placing it in a buffer and then waiting for an interrupt to indicate that the character has been transmitted.

C. Design

The design presented here, for consistency with the Nielsen–Shumate and Sanden approaches, assumes a concurrent Ada environment. The object-oriented model is mapped to an Ada design in a manner similar to that described by Booch [8]. Each class is designed as an Ada package. Active objects, those which can exhibit an independent thread of behavior, include a task inside the package which controls that behavior. Procedures exported by the package encapsulate the task's rendezvous. The design is represented graphically using the notation developed by Buhr *et al.* [10].[2]

The design for the dominant domain is shown in Fig. 1. There are three packages representing the classes FURNACE, THERMOMETER, and RTS_HOST_INTERFACE. Since there are 16 instances of the class FURNACE, each of which has an independent thread of behavior, this package includes an array of tasks. Each task controls the operation of one FURNACE instance. The packages for THERMOMETER and RTS_HOST_INTERFACE indicate their interfaces to FURNACE. Internal details of their operation belong to supporting domains, not the dominant domain, and are not represented in this figure.

Fig. 2 illustrates the design for the DC_UTIL. This design contains two packages: ADC and SENSOR_SPEC. Since the interaction with the ADC can be an independent thread, it is assigned to a task. The task both serializes access to the ADC and provides an interrupt handler via a rendezvous

[2]The figure presents a schematic view of the design. Additional details may be found in [60].

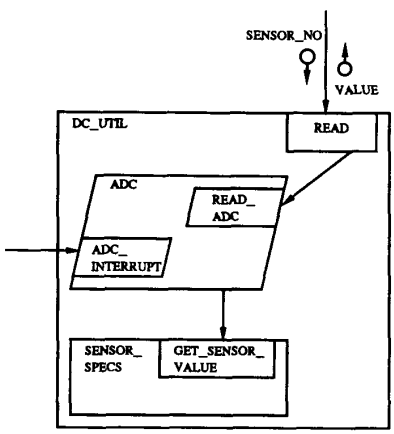


Fig. 2. Data capture utility design.

assigned to the interrupt address. The SENSOR_SPEC class does not require a task. In addition, since SENSOR_SPEC is used only by the DC_UTIL, it is encapsulated within this package. This is a generic package. The parameters used to instantiate the generic are the ADC interrupt, sensor, and buffer addresses (ADC_INTERRUPT_ADDR, ADC_SENSOR_ADDR, and ADC_BUFFER_ADDR) and the type definitions for SENSOR_IDs and VALUES.

The design for the generic COMMS_UTIL, shown in Fig. 3, is obtained by following a similar procedure. Each class is designed as a package with an internal task which controls the behavior of the class. The packages for the INPUT_MESSAGE, OUTPUT_MESSAGE, INPUT_PORT, and OUTPUT_PORT classes are all contained within the generic COMMS_UTIL package. This package is instantiated by providing appropriate addresses for the input and output ports and type definitions for INPUT_MESSAGE and OUTPUT_MESSAGE.

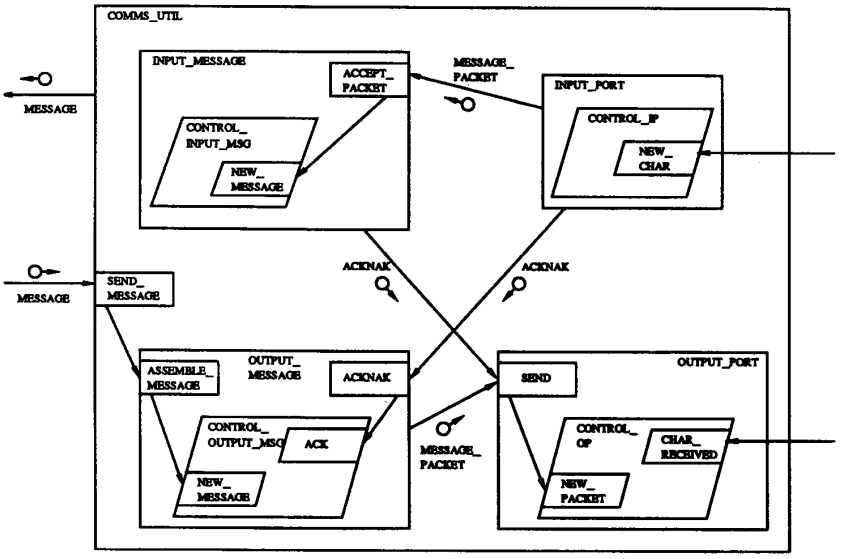


Fig. 3. Communications utility design.

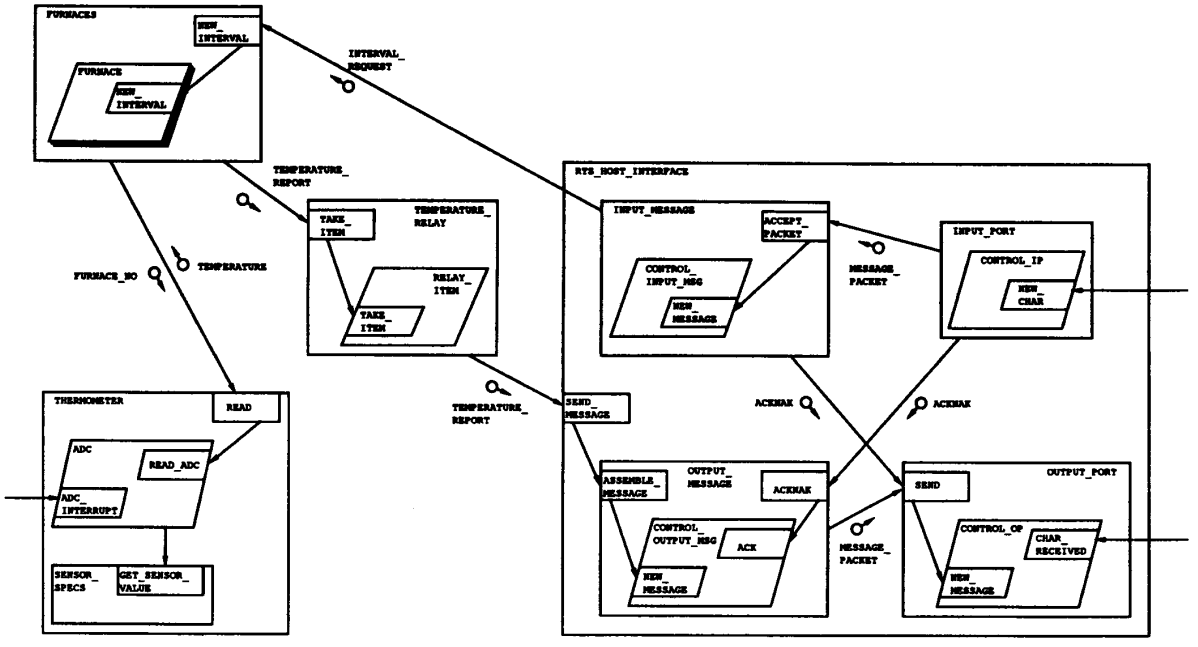
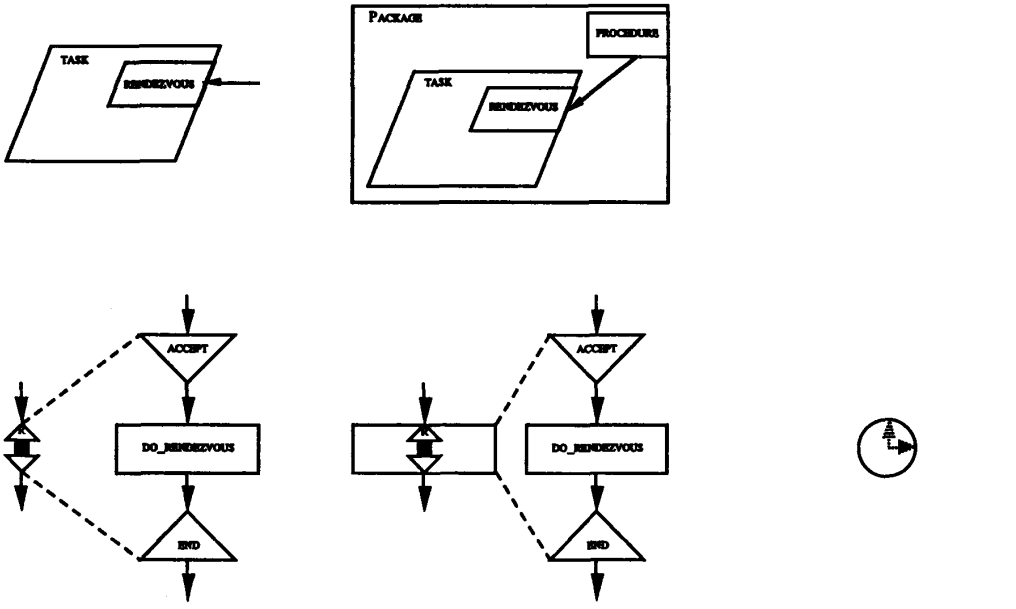


Fig. 4. Integrated RTS design.

Fig. 4 shows the integrated design with names on the package interfaces that reflect the generic instantiations. The integrated design contains an additional class, TEMPERATURE_RELAY. The purpose of this class is to decouple the communication between the instances of FURNACE and OUTPUT_MESSAGE. The reason for introducing this class is by no means obvious from the problem description alone. The need for its inclusion was revealed by the SPE analysis of a preliminary version of the design. Details of this analysis are in Section V. The TEMPERATURE_RELAY task is discussed in Section VI.

V. SPE Models

Having developed a preliminary design, we are now in a position to evaluate its performance characteristics. If no



(a) Rendezvous entry calls, accept, and end-accept nodes

(b) Shorthand notation for encapsulated rendezvous

(c) Processing delay node

Fig. 5. Real-time extensions to execution graphs.

problems are indicated, we can proceed to the next phase of the development process. If there are problems, they must be corrected before proceeding.

In order to analyze the design, it was necessary to extend the execution graph notation described in [53] to add nodes to represent rendezvous entry calls, accept, and end-accept structures. They are illustrated in Fig. 5(a). The execution graph node in Fig. 5(b) represents the special case when a procedure call (only) hides a rendezvous entry call and the procedure does no other significant processing. The symbol in Fig. 5(c) represents a delay in processing.

A. Case Study Execution Graphs

The first step in SPE modeling is to identify the significant behavior patterns of the system. These come directly from the class behavior pattern models constructed during the analysis/design phase. As described in Section IV-B, the system begins in an IDLE state. When an INTERVAL_REQUEST is received, the THERMOMETER is read. When the reading is complete, each FURNACE task enters a loop in which it waits for the time interval to lapse, or a INTERVAL_REQUEST to arrive, then repeats these steps as long as the system operates. The FURNACE execution graph in Fig. 6(a) represents this part of the scenario.

The number of external entities (FURNACEs = 16) and the time between events specifies the intensity of events in this scenario. Temperature readings [represented by the READ_THERMOMETER expanded node in Fig. 6(a)] occur every 10–99 s for each FURNACE. Therefore, this design must be capable of processing one READ_THERMOMETER event per FURNACE per 10 s. The frequency of INTERVAL_REQUEST events is not specified, but it is reasonable to assume that they are infrequent relative to the temperature readings and that the system stabilizes into steady-state behavior between INTERVAL_REQUESTs. The SPE analysis strategies lead to an initial model of this steady-state behavior without INTERVAL_REQUESTs for an initial check of the design. Later, after any problems are resolved, the model can incorporate the INTERVAL_REQUEST behavior.

The execution graph in Fig. 6(b) elaborates the behavior of READ_THERMOMETER. After delaying until the TIME interval lapses, it calls THERMOMETER.READ then reports the temperature (at TEMPERATURE_RELAY.TAKE_ITEM) and sets the next TIME delay. Figure 4 shows that THERMOMETER.READ hides the rendezvous at READ_ADC and does no other processing. Therefore the THERMOMETER.READ symbol in Figure 6(b) is the shorthand notation from Figure 5(b). Similarly TEMPERATURE_RELAY.TAKE_ITEM hides the rendezvous at RELAY_ITEM.TAKE_ITEM.

The behavior of the expanded node READ_ADC is elaborated in Fig. 6(c). It ACCEPTS the READ_ADC entry call, selects the sensor for the appropriate furnace, then waits on an ACCEPT for an interrupt from the ADC that signals that the reading is available. When the interrupt occurs, processing continues to convert the ADC reading, end the interrupt rendezvous, and release the FURNACE that initiated

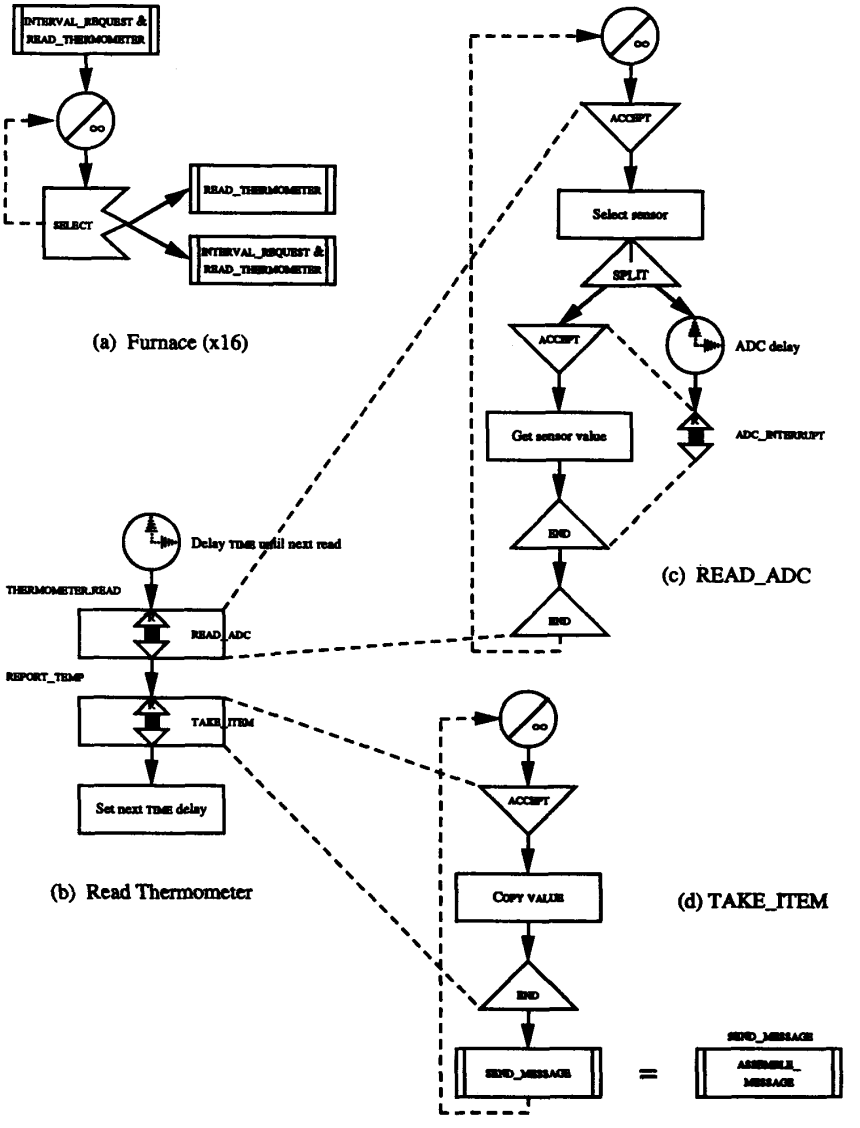


Fig. 6. Scenario execution graph.

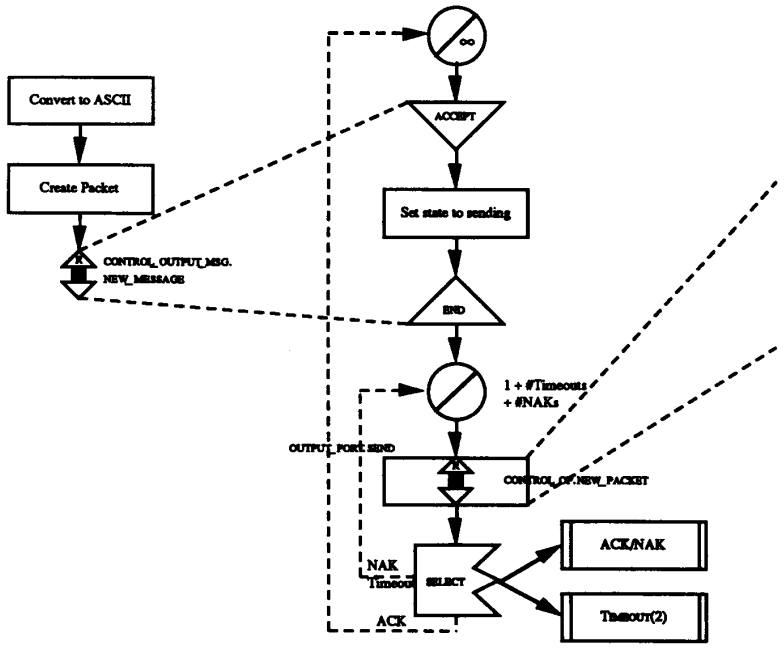
the reading (indicated by the END nodes). READ_ADC then returns to the ACCEPT node to await another entry call. Note that the task ADC waits on the asynchronous ADC device. Execution graphs usually represent asynchronous tasks with separate graphs; however, with this design, the ADC device is controlled exclusively by the task ADC. The split node shows where the ADC device is activated, the delay for the sensor to settle, and the ADC_INTERRUPT rendezvous to signify completion.[3]

Fig. 6(d)–(k) illustrate the remaining execution graph processing for the case study.

[3] Note that so far, the graphs represent only behavior patterns; the allocation of tasks to processors is not yet specified. Eventually, the models will show that the ADC does not consume resources on the RTS processor.

B. Scenario Data

The next SPE step is to collect data on the execution environment and resource usage estimates for each operation in the execution graph models. The most significant hardware devices in the system are the RTS processor, the ADC device, and the host communication channel. The initial model represents a single processor system; a bounds analysis evaluates a range of processor power. The most significant software service routines are the rendezvous and procedure call overhead. The operations in the execution graph scenario at this stage in development are simple. Following the SPE best-case analysis strategy, the initial models ignore internal processing for operations, and represent only the overhead for invoking them.



(e) ASSEMBLE_MESSAGE (f) CONTROL_OUTPUT_MSG.NEW_MESSAGE

Fig. 6. *(Continued).*

Table I shows the estimated average service time for each of the hardware devices and representative service requirements for procedure call and rendezvous overhead. The CPU requirements for Ada procedure calls and rendezvous are taken from [11], [24], and [43]. Service times for the ADC and communications line are estimated.[4]

The execution graph for the furnace scenario has both a repetition loop and a conditional path; however, in this steady-state scenario only the READ_THERMOMETER conditional path in the inner loop executes. It is an expanded node, so the total R_i for the READ_THERMOMETER subgraph is the total for the scenario. The repetition loop executes once per temperature reading. In this steady-state scenario, we want the resource requirements within the loop. We will use the result in the system execution model to evaluate the effect of multiple furnace tasks and their time interval between readings. Similarly, we set the repetition factor to 1 in the graphs for READ_ADC, TAKE_ITEM, and all other tasks that make one loop repetition per reading.

Let

1 = number of procedure calls
2 = number of rendezvous entry calls
3 = number of ADC activations
4 = number of Host communication transmissions
5 = number of temperature readings sent to host
6 = number of time interval delays

then $(r_1, \cdots, r_6)$ is the requests per node, and $(R_1, \cdots, R_6)$ is the requests per graph. Table II shows the number of procedure calls, rendezvous, and requests for hardware device service in each execution graph. Following the simple-to-realistic strategy, initial models attribute all of the rendezvous overhead to the operation making the entry call. Later, if deemed appropriate, models could prorate the overhead between calling and called tasks (if data exists that distinguishes between entry call and accept overheads). The following explains the derivation of the values in Table II.

The nodes in the READ_THERMOMETER subgraph (and most of the other graphs) form a sequential path. The sequential path reduction rule sums the r_i for each node to produce the result for item (b) of Table II. A best-case assumption for the CONTROL_OUTPUT_MSG.NEW_MESSAGE graph [Fig. 6(f)], is that neither timeouts nor NAKs occur. Therefore the loop repetition factor = 1, the probability for the ACK/NAK path is 1, and the probability of the Timeout path is 0. The derivation of the results in Table II for the other graphs is straightforward.

Next, consider the SPLIT nodes in the READ_ADC graph and the CONTROL_OP.NEW_PACKET graph [Fig. 6(c) and (g)]. In both cases, the task executing on the Remote Temperature Sensor (RTS) processor issues an ACCEPT and waits for a response from an external entity. A simple model of this

[4] These numbers represent reasonable initial estimates. The SPE models indicate that the performance of the RTS system is not particularly sensitive to their magnitude and the estimates are not refined further for later models. If sensitivity studies had indicated that these values were crucial, more refined estimates or measurements would have been obtained.

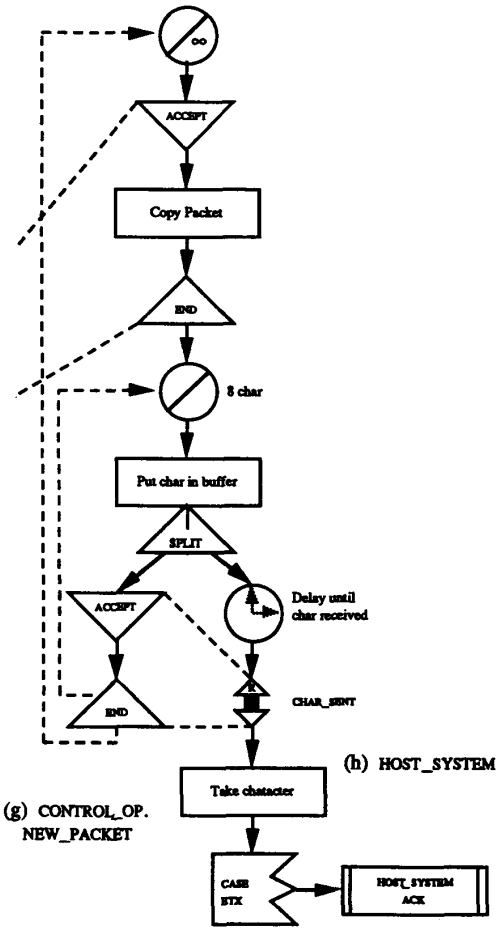


Fig. 6. *(Continued).*

behavior represents the number of external interactions (r_4 and r_5) and an estimate of the average delay time for each. Later, studies can examine the sensitivity of RTS performance to external delays. The estimated ADC delay is the average service time in Table I. The estimated HOST_SYSTEM "Delay until character received" is 0 (in the best case). The other processing in the HOST_SYSTEM graph, and any other (unspecified) processing and delays that occur before the host transmits the ACK, combines into "Delay until ACK," with an estimated delay ranging between 10 and 100 ms.

Next, we compute a lower bound on processing time T_j for each graph j

$$T_j = \sum_{i=1}^{k} R_i w_i$$

where

$w_i \quad i = 1, \cdots, k$ is the average service time per request for resource i.

The last two columns in Table II show the T_j values for the case study for the lower and upper bound values for w_i (in the first two rows of the table). The lower bound on total processing time for a temperature reading, T, in this example is the sum for all execution graphs j,

$$T = \sum T_j.$$

The upper and lower bounds for T are the total for the corresponding column in Table II. Clearly, the system can process one reading for one furnace in 10 s for both lower and upper bounds on device service and estimated delays. Next a more realistic system execution model represents contention delays among the 16 furnace tasks.

C. System Execution Model

Fig. 7 shows a system execution model in which a task begins executing on the RTS processor. Upon completion (of a specified amount) of service it makes a request of the ADC (represented by a delay server). After the specified delay, it returns to the RTS. Upon completion, it visits the HostComms, returns to the RTS, and repeats this pattern until it has sent eight packets, then it visits ACKDelay. After the specified delay it returns to the RTS where it completes

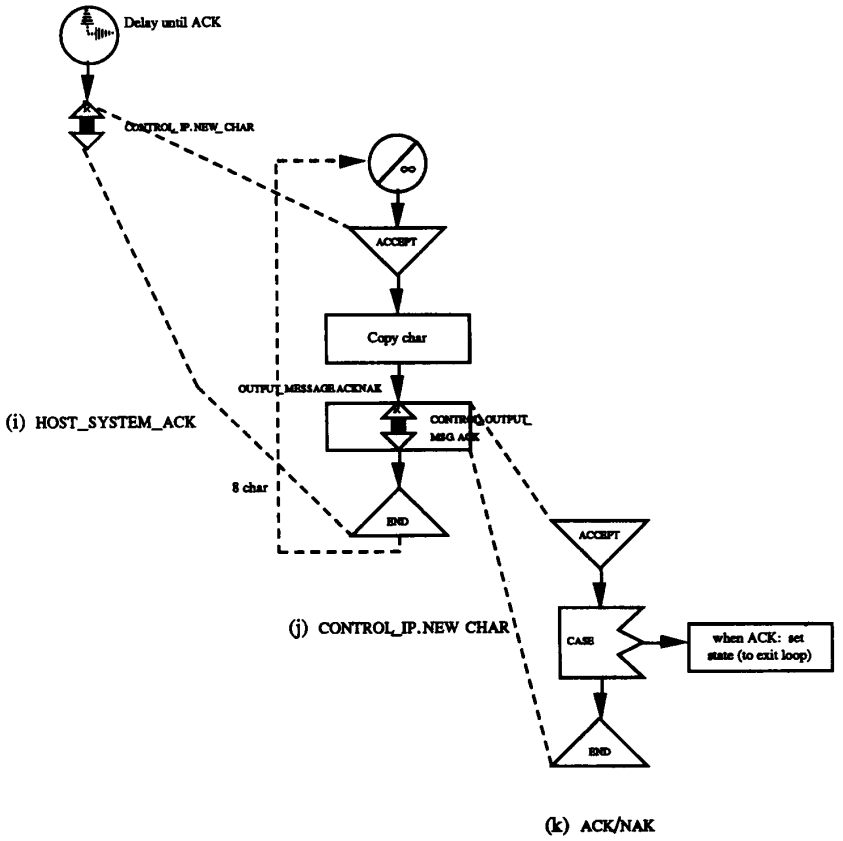


Fig. 6. *(Continued).*

TABLE I
ESTIMATED AVERAGE SERVICE TIME

Hardware Devices	Estimated Average Service Time
ADC	25 ms.
Communications Line	1 ms./packet

Software Resources	Estimated Average CPU Time
Procedure Calls	.0110 - .0201 ms
Rendezvous	.5030 - 1.0010 ms

Execution Delays	Estimated Average Delays
Time to ACK each character	0 sec
Time to ACK data packet	0.010 - 0.100 sec
Time between temperature readings on each furnace	10 sec

the read temperature processing, then goes to the IntTimer. After the specified timer delay it returns to the RTS to repeat the above process for another temperature reading. This model includes rendezvous processing but ignores potential rendezvous synchronization delays between tasks; they are added later in the advanced system execution model.

The parameter values come from the data collected for the software execution model. The initial system execution model combines all processing into one transaction category representing Furnace tasks. The workload intensity, M, for the Furnace category is 16. The execution graph results specify the service requests and service time. In this example, the visits to the ADC, HostComms, ACKDelay, and IntTimer are R_3 through R_6 respectively; and their service times are w_3 through w_6. In this example, the flow through the queue-servers described above routes Furnace tasks through the RTS after each visit to another queue-server. Therefore the visits to the RTS is the sum of the visits to other devices,

$$V_{RTS} = \sum_{i=3}^{6} R_i.$$

The average RTS service time is the total RTS demand, D,

TABLE II
EXECUTION GRAPH RESOURCE REQUIREMENTS, R_i

Execution graphs	Proc calls	Rendezvous	ADC visits	HostComms	ACK delay	IntTimer	Lower Bound Tj	Upper Bound Tj
Lower bound	0.0000110	0.000503	0.025	0.001	0.01	10		
Upper bound	0.0000201	0.001001	0.025	0.001	0.1	10		
(a) Furnace (expanded in (b))								
(b) Read thermometer	2	2				1	10.001	10.002
(c) Read ADC	2	1	1				0.026	0.026
(d) Take item	2						0.000	0.000
(e) Assemble message	1	1					0.001	0.001
(f) Control_output_msg. new	1	1					0.001	0.001
(g) Control_output.new_packet		8		8			0.012	0.016
(h) Host_system							0.000	0.000
(i) Host_system ack					1		0.010	0.100
(j) Control_IP.New char	1	1					0.001	0.001
(k) Ack/Nak							0.000	0.000
TOTAL	9.000	14.000	1.000	8.000	1.000	1.000	10.050	10.147

TABLE III
SYSTEM EXECUTION MODEL PARAMETERS

Workload: Furnace
Workload intensity: 16 tasks

Nodes	Service requests, Vi	Service time, Si (sec)
ADC	1	0.025
HostComms	8	0.001
ACKDelay	1	0.01 - 0.10
IntTimer	1	10.0
RTS	11	0.000649 - 0.0013

averaged across all visits:

$$D_{RTS} = \sum_{i=1}^{2} R_i w_i$$

$$S_{RTS} = \frac{D_{RTS}}{V_{RTS}}.$$

Table III shows the model parameters for the case study. These formulas apply specifically to this case study. The general algorithms for transforming software execution models into system execution model parameters are in [53].

The model results for the case study show that the proposed system can process one reading per furnace for 16 furnaces with an average response time of 57.7 ms for the upper bound specifications (when there are no significant rendezvous synchronization delays). In fact, the processor may be oversized for this system.[5] Under more realistic circumstances one might, for example, use the excess capacity to perform additional functions on this processor. The system execution model can quantify the interaction among functions: represent the additional functions with transaction categories, specify model parameters for each, solve the model, and examine the results by category. Perhaps the best-case service requirements and other processing functions greatly underestimate the actual resource requirements. As better data becomes available, SPE adapts the models to examine upper bounds, worst-case assumptions, and NEW_INTERVAL requests, and vary the RTS processor size (service rate) to forecast actual capacity requirements.

[5] The processor "size" used here was determined by the availability of data for Ada procedure calls and rendezvous.

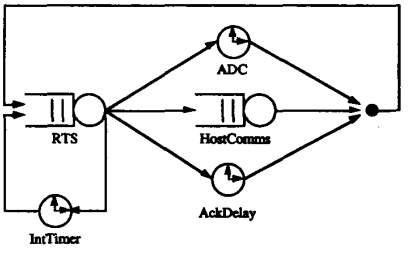


Fig. 7. Initial system execution model.

The next step is to augment the system execution model to predict the effect of rendezvous between concurrent tasks. There are some approximate analytic methods that quantify the average delay to establish a rendezvous when, for example, 16 FURNACE tasks make entry calls to 1 RELAY_ITEM task. See [46], [63], [62], [10] for specific algorithms. In this case study, those models show that the *average* values for rendezvous delay are insignificant and the models are not described here.

Even though average delays are unlikely to be a problem, there may be periods of time during normal operation when many furnace tasks request readings in a short period of time (i.e., "burst" behavior). To examine details of synchronization

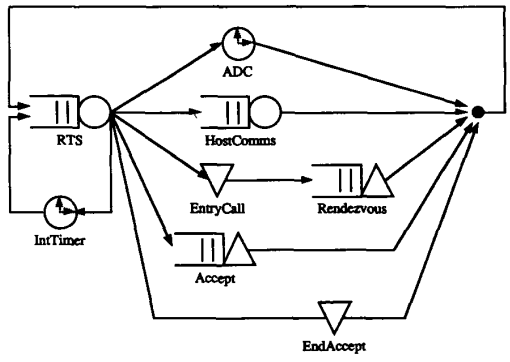


Fig. 8. Rendezvous system execution model.

we use advanced system execution models with real time extensions to represent: entry calls, rendezvous completions, accepts, and end accepts. Distinct transaction categories represent the tasks that synchronize. The extensions preclude approximate analytic solutions; they require simulation techniques. The following section explains the modeling process with the case study example.

D. Rendezvous System Execution Model

The real time extensions to represent the Ada rendezvous use IPG ALLOCATE and RELEASE nodes to represent rendezvous synchronization [41]. IPG's use *tokens* to represent software resources; there is a distinct *token type* for each resource modeled. ALLOCATE nodes grant tokens to transactions. A transaction visits an allocate node and requests a specified number of tokens of various types. If they are not available, the transaction waits in the queue. The transaction leaves the allocate node when the request is granted. It keeps the tokens until it reaches a RELEASE node. RELEASE nodes take the specified number of tokens of various types from the transaction and give them to the designated allocate node.

Fig. 8 shows the rendezvous system execution model. The Accept and Rendezvous nodes are ALLOCATE nodes. The EntryCall and EndAccept are RELEASE nodes. The execution of a Furnace transaction category in Fig. 8 is as follows. Each Furnace transaction starts with a token representing each type of rendezvous it will make: one ReadTherm for the rendezvous with READ_ADC and one RepTemp for the rendezvous with TAKE_ITEM. The Furnace transaction executes on the RTS until ready to rendezvous with READ_ADC. Furnace then goes to the RELEASE node representing the EntryCall. It sends its ReadTherm token to the Accept node. Furnace then proceeds to the Rendezvous ALLOCATE node where it waits for the ReadTherm token to be returned. Furnace receives the ReadTherm token when the corresponding EndAccept statement releases the caller. Then Furnace returns to the RTS, executes, then returns to the EntryCall node, sends its RepTemp token to the Accept node, goes to the Rendezvous node, and waits for the RepTemp token to be returned. When Furnace is allocated the RepTemp token, it returns to the RTS completes processing then goes to the IntTimer to delay until the next reading.

Another transaction category, Therm, represents the READ_ADC execution graph. It begins at the Accept node where it waits for a ReadTherm token. When the token arrives it goes to the RTS, executes, then goes to the ADC, delays, returns to the RTS, executes, then goes to the EndAccept where it releases the ReadTherm token to the Rendezvous node. It returns to the RTS, completes processing, then returns to the Accept where it either waits for another ReadTherm or begins processing if one has already arrived.

These models could represent each of the execution graphs in Fig. 6(a)–(k) with a separate transaction category and route them through the allocate and release nodes as appropriate. However, there are two compelling reasons to use a different strategy: 1) the complexity of the resulting model makes it difficult to study, interpret, and explain results; and 2) the duration of the simulation run greatly increases with the number of transaction categories and their processing complexity, thus limiting the number of SPE model studies.

A graph reduction step identifies the rendezvous dependencies that may introduce significant delays in the processing. For example, the graph in Fig. 6(f) begins with an Accept, and releases the caller after a minimal amount of processing. However, the rest of the processing in the graph must complete before the task will return to Accept another request. Even though the calling task is released to make another request, subsequent requests will not be processed until CONTROL_OUTPUT_MESSAGE.NEW_MESSAGE has finished the previous request. Thus, the reduction step identifies the subordinate processing executed by subordinate tasks and aggregates it into transaction categories. All the rendezvous overhead is in the resulting composite task, but "internal" rendezvous do not visit the ALLOCATE and RELEASE nodes.

The graph reduction step reduces the Remote Temperature Sensor model to three transaction categories: Furnace [item (b) in Table II], Therm [item (c)], and TReport [items (d-k)]. The model parameters are in Table IV. The results of the simulation are in Table V.

VI. Discussion

Under ideal conditions, one would use SPE to help evaluate design alternatives. The existence of previous solutions to the RTS problem provides that opportunity here. In the following sections we describe the designs obtained by Nielsen and Shumate and Sanden and compare their properties.

A. Design Comparisons

Solutions to the Remote Temperature Sensor problem have also been presented by Nielsen and Shumate [42] and Sanden [48]. The design presented by Nielsen and Shumate is based on traditional structured analysis techniques, augmented with concepts from Pamela [13] and [9]. The technique involves an "edges-in" functional decomposition. Their design is shown in Fig. 9.

Sanden's design is based on an "entity-life modeling" approach which identifies threads of events corresponding to life histories or behavior patterns associated with real-world entities in the application. Threads are then modeled using one

TABLE IV
PARAMETERS—RENDEZVOUS SYSTEM EXECUTION MODEL

Transaction categories	Service Requests, Vi			Service Time, Si		
	Furnace	Thermo-meter	Temp Report	Furnace	Thermo-meter	Temp Report
Nodes						
ADC		1			0.025	
HostComms			8			0.001
ACKDelay			1			0.010
IntTimer	1			10.0		
EntryCall - Rendezvous	2			N/A		
Accept		1	1		N/A	N/A
EndAccept		1	1		N/A	N/A
RTS- lower	3	3	11	0.00034	0.00018	0.00051
RTS - upper	3	3	22	0.00668	0.00035	0.00101
Total RTS Demand - lower				0.00102	0.00054	0.00561
Total RTS Demand - upper				0.00204	0.00105	0.0111

TABLE V
RESULTS—RENDEZVOUS SYSTEM EXECUTION MODEL

Workload	Response Time (ms.)		RTS % Utilization
	Average	Maximum	
Furnace	37.51	186	0.3
Thermometer	25.9	173	0.2
TempReport	30.7	104	1.8
TOTAL	94.1	463	2.3

(Note: The RTS device has the maximum utilization in the model, others are not shown)

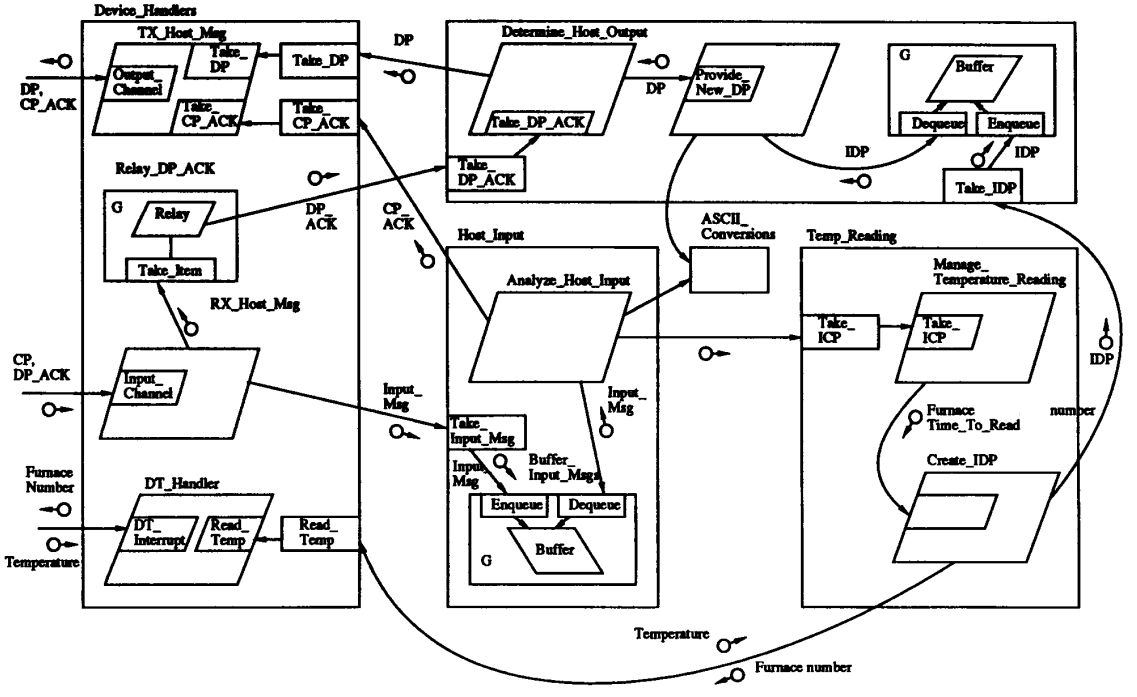


Fig. 9. Nielsen–Shumate RTS design.

of a variety of notations: JSD [25], state-transition diagrams, or Statecharts [21]. In the design, each thread is mapped onto a task. Fig. 10 illustrates Sanden's design for the RTS problem.

The approach to domain modeling used here is conceptually closer to Sanden's entity-life modeling technique than to the functional approach used by Nielsen and Shumate.[6] The identification of real-world entities in the application and

[6]The object-oriented approach described in [60] does, however, provide a more systematic approach to identification of entities and modeling of their operations than is evident from the description of entity-life modeling in [48].

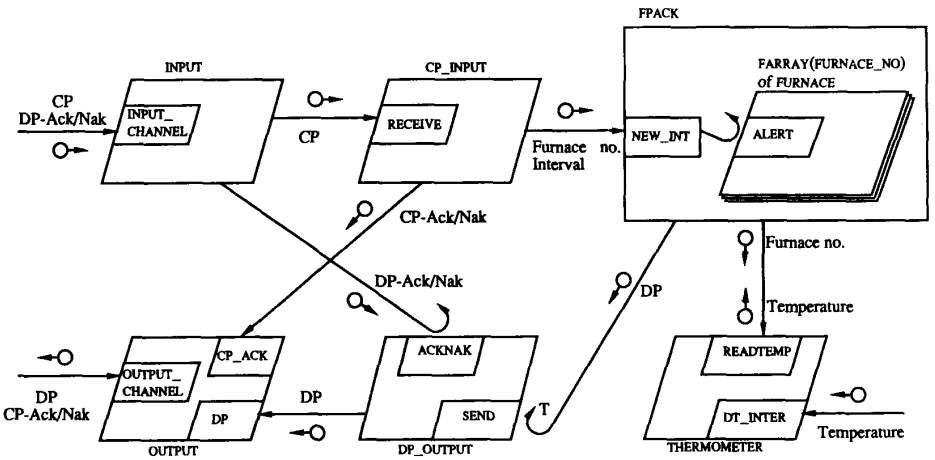


Fig. 10. Sanden RTS design.

modeling of their life histories provides some of the elements of an object-oriented approach. Sanden's entity life-histories are, for example, analogous to the class behavior pattern models used here. Thus, while our approach uses different heuristics than Sanden's, it is not surprising that the tasking structure shown in Fig. 4 is similar to that presented in Fig. 10.

Sanden provides a comparison of his design and the one presented by Nielsen and Shumate in terms of number of tasks, number of rendezvous, and understandability. The entity-life approach produces a design that has more tasks than the functional approach but is conceptually simpler since the larger number of tasks is due to the existence of identical tasks for each of the 16 furnaces. The entity-life approach also contains fewer rendezvous which, according to Sanden, produces a more efficient design.

Since one of the goals of domain analysis is enhanced reuse, it is also useful to compare the various remote temperature sensor designs along this dimension. The functional approach used by Nielsen and Shumate produces components with relatively low potential for reuse. The functionality of the various components is very application-specific. In addition, while the functional decomposition groups components logically for this application, the grouping is unlikely to be reusable in others. This problem is typified by their package DEVICE_HANDLERS which includes the device interfaces for the digital thermometer as well as communication with the host.

The entity-life modeling design does not represent a significant improvement in terms of reuse. The tasks which perform host input/output or which interface with the digital thermometer are still application specific, as in Nielsen and Shumate. The packaging of the design also does not lend itself to reuse since these tasks cannot be stand-alone compilation units.

The design produced via domain analysis, while structurally similar to that produced via entity-life modeling provides substantial opportunities for reuse. As noted above, both the DC_UTIL and the COMMS_UTIL are generic components which are potentially reusable across many different applications. FURNACE components are also potentially reusable in related applications within the domain of process control. The domain analysis led to explicit consideration of model integration issues, enhancing the ease with which these components can be reused. We also note that this level of reuse was achieved without sacrificing the conceptual simplicity of the design.

Performance analysis of the Nielsen/Shumate and Sanden designs reveals some interesting comparisons. As noted above, Sanden claimed that his design was "more efficient" based on the number of rendezvous required to make and report a temperature reading (four versus eight for the Nielsen and Shumate design). These comparisons are discussed in the next section.

B. Performance Comparisons

To compare the performance of the three designs, we first formulate execution graphs for each design. Figs. 11 and 12 show the most significant parts of the Nielsen–Shumate and Sanden execution graphs. The extended software execution graph analysis using upper bound specifications yields the results in Table VI. All designs have acceptable performance when there are no contention delays. The second column in Table VI shows the results of the next step; the system execution model analysis. Both the Sanden design and the one presented here have 16 concurrent Furnace tasks. The Nielsen–Shumate design has only one task. Again, there are no evident performance problems in any of the designs.

To compare the results in the extended system execution model, the model formalism needs an additional extension to represent the buffer synchronization in the Nielsen–Shumate design. The extension is defined in the Appendix.

The workload reduction step yields three workloads for each of the designs; however the processing within each workload differs in each model. The results of the three rendezvous system execution models are in Table VII. Again, all designs

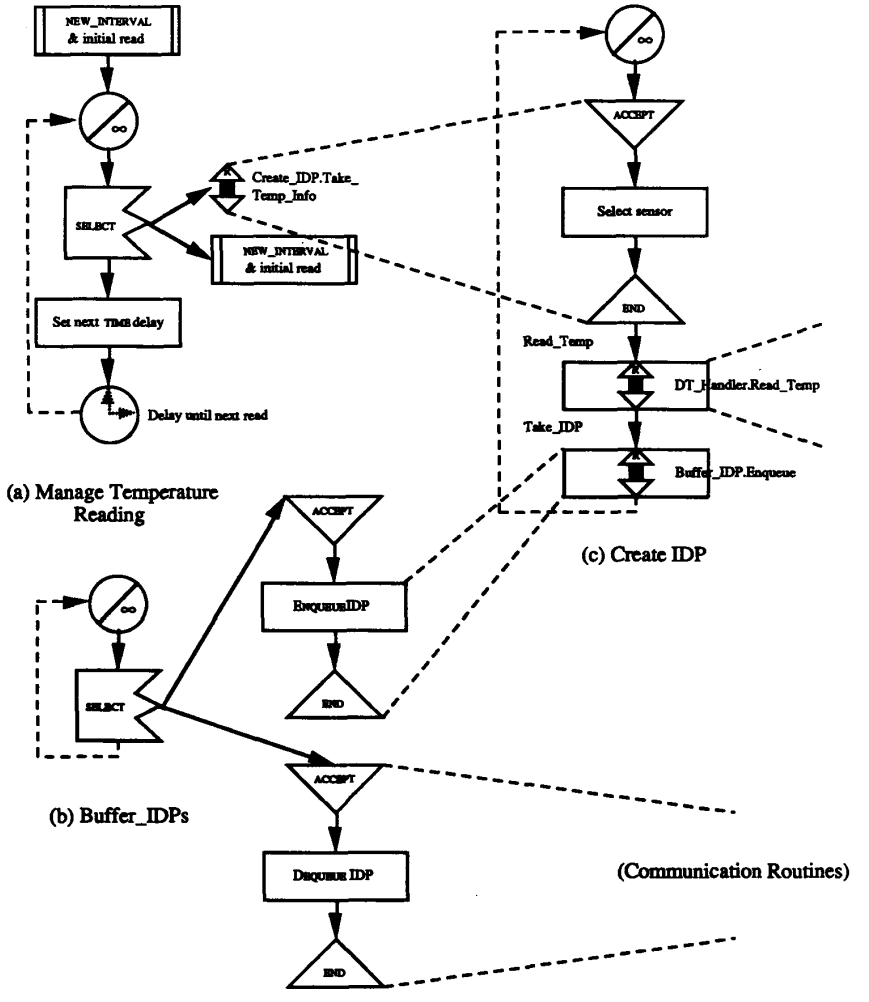


Fig. 11. Partial Nielsen–Shumate scenario.

have acceptable response times. It is, however, evident that both our design and Sanden's incur a larger (though still acceptable) overhead due to contention delays. Thus, Sanden's claim that his design is "more efficient" is not supported.

Note the results for the average queue length at the Rendezvous node. While the average queue length is low, both our design and Sanden's models show that, at some time during the simulation, there were up to four tasks waiting to rendezvous with the Thermometer workload and up to three tasks waiting to rendezvous with the workload that sends the data packet. This observation led to the introduction of the TEMPERATURE_RELAY task in Fig. 4. This task decouples the FURNACE tasks from the communications and allows each FURNACE to proceed after forwarding its TEMPERATURE_REPORT. The SPE model provides the data necessary to quantify the size of the buffer required in TEMPERATURE_RELAY.

Sanden's design addresses this problem using a timed entry call. Although there is a potential for the rendezvous to time out with a resulting loss of data, the SPE analysis shows that this is not a problem under the specified conditions.

Based on the design evaluation and performance analysis, developers can now choose from among the candidate designs. Since all of the candidate designs appear to provide adequate performance, the choice can be based on other criteria (e.g., reusability).

To illustrate how SPE might provide additional support for the design process, consider what would happen if the rendezvous at SEND in Fig. 10 were conditional instead of timed. A conditional rendezvous succeeds if the callee is ready to Accept, otherwise the caller continues its execution without delay. The model extension checks the queue length at the Accept node before the GetTemps workload goes to the EntryCall model node. If the SendDP transaction is currently at the Accept node, GetTemps proceeds to the EntryCall node as usual. Otherwise, the rendezvous fails and GetTemps continues its execution without visiting the EntryCall or the Rendezvous nodes.

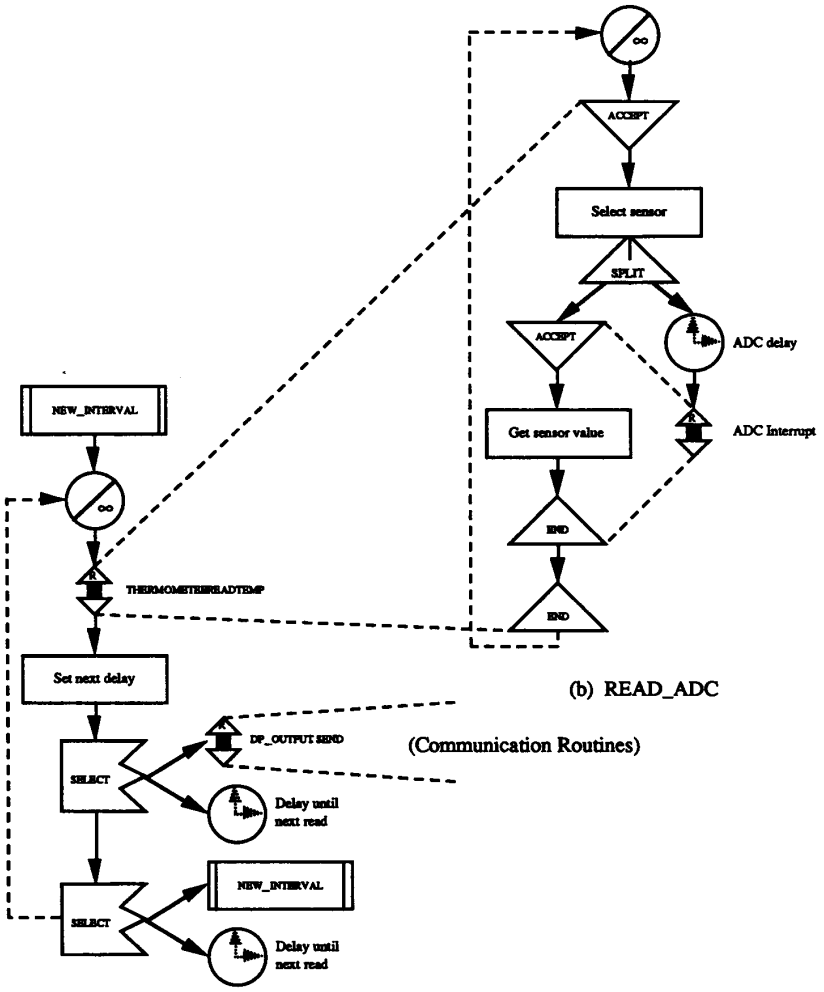


Fig. 12. Partial Sanden scenario.

The results of a conditional rendezvous model for the Sanden design are shown in Table VIII. It shows that out of 960 attempted rendezvous (in a 600 s modeled interval), 244 rendezvous would fail! Because the tasks are periodic (with a deterministic delay between readings) the system with conditional rendezvous repeatedly drops temperature readings for the same furnace. This result is based on very optimistic assumptions. It assumes that the temperature-reading time for the 16 furnaces is uniformly spread across the 10 s interval (to minimize the probability of collisions) and uses lower bound delays for both character ACKs and temperature reading ACKs. A sensitivity study shows that relaxing these assumptions greatly increases the number of dropped readings.

Thus, the real time extensions to the system execution models not only quantify performance of various designs, they can also identify potential design errors early in the development process, before coding begins.

TABLE VI
INITIAL DESIGN COMPARISON

Designs	Minimum Response Time (ms.)	System Execution Model Response Time (ms.)
Nielsen-Shumate	61	61.0
Sanden	56	56.4
Current Design	57	57.7

VII. CONCLUSIONS

Software Performance Engineering is an important and often neglected aspect of systems development. Including explicit consideration of performance issues can help avoid problems such as cost overruns, missed deadlines, and unmet performance goals. Our experience indicates that SPE can help eliminate the need for post-implementation tuning that increases cost and decreases maintainability.

The case study presented in Sections IV and V illustrates

TABLE VII
RENDEZVOUS SYSTEM EXECUTION MODEL COMPARISONS

	Response Time (ms)		Rendezvous Queue Length		Waiting on:	RTS
	Avg.	Max.	Avg.	Max		% Util.
Nielsen-Shumate:						
Manage Temp	29.3	172	0	1	BufferIDPs	0.7
Buffer IDPs	1.4	12				0
SendDP	34.0	95	0.946	1	BufferIDPs	2.2
TOTAL	64.7	279				
Sanden:						
GetTemps	37.0	158	0.046	4	Thermometer	0.3
			0.01	3	SendDP	
Thermometer	25.3	149				0.2
SendDP	29.8	103				1.6
TOTAL	92.1	410				
Current Design:						
Furnace	37.5	186	0.05	4	Thermometer	0.3
			0.01	3	TempReport	
Thermometer	25.9	173				0.2
Temp Report	30.7	104				1.8
TOTAL	94.1	463				

TABLE VIII
CONDITIONAL RENDEZVOUS SYSTEM EXECUTION MODEL RESULTS

	Response Time (ms)		Rendezvous Queue Length		RTS
	Avg.	Max.	Avg.	Max	% Util.
GetTemps	43.1	229	0.06	5	0.3
			0.002	1	
Thermometer	28.2	163			0.2
SendDP	29.5	85			1.2
TOTAL	100.8	477			
SendDP Rend. Tried	960				
SendDP Rend. Failed	244				

how Software Performance Engineering may be applied to the design phase of systems with real-time/reactive characteristics. The results of the case study indicate that it is possible to achieve adequate performance with a design that is conceptually simple, supports reuse, and is easy to modify. The results also indicate that simplistic metrics, such as the number of rendezvous are not sufficient to compare designs for real-time systems.

In fact, SPE provides support for all phases of the development process from specification through implementation. During specification, SPE can help assess the reasonableness of system requirements. As we have seen, during design, SPE provides techniques for assessing the performance properties of designs and can assist in selecting from among design alternatives. During the implementation phase, SPE helps ensure that performance goals are met. Information obtained from SPE can also be used to guide the development process, for example by identifying critical components for prototyping.

While this paper has illustrated the use of SPE with a particular method and notation, SPE techniques can be easily integrated with a variety of development approaches. SPE is a systematic method for applying quantitative methods independent of the specific methods, notations, and even modeling tools. Execution graphs and IPG's serve as the basis for using multiple modeling techniques and tools. The models evolve with the software and progress from simple, optimistic models to more detailed, realistic models. They can evolve beyond the queueing network model simulations described here to automatically generate Petri net models for more intricate analyses of implementation details.

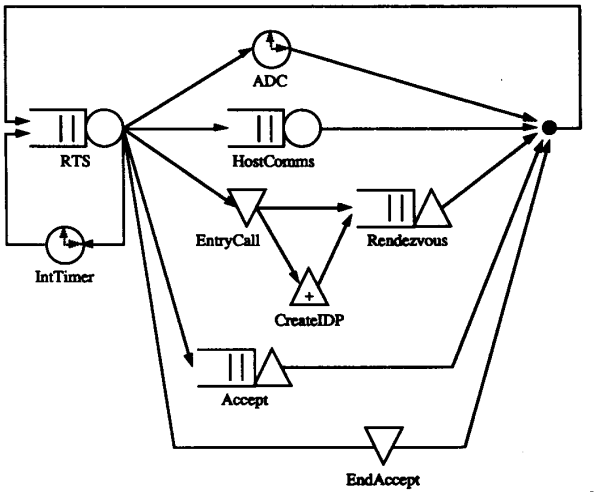


Fig. 13. Buffer model extensions.

APPENDIX

A. Buffer Model Extensions

The Nielsen–Shumate model has a buffer workload we call BufIDP. The GetTemps workload makes an entry call to BufIDP to enqueue a data packet (IDP). The SendDP workload makes an entry call to BufIDP to dequeue a data packet. The rendezvous system execution model in Section V ensures that the SendDP cannot proceed until the BufIDP dequeue end-accept completes; however it would let the SendDP workload proceed even if no IDP's were enqueued. The model in this section allows the SendDP rendezvous to complete only when at least one IDP is in the buffer and the BufIDP end-accept

completes. The model assumes the buffer size is infinite—the model results report the average and maximum IDP's in the buffer, so designers learn how large the buffer needs to be.

The buffer model is in Fig. 13. It adds a CREATE IPG node, CreateIDP. The GetTemps workload visits EntryCall and releases its IDPAck token to the Accept node. Then it visits the CreateIDP node where it creates an IDP token and sends it to Accept. GetTemps proceeds to the Rendezvous node to wait for its IDPAck token to be released by BufIDP.

When the BufIDP workload accepts an IDP token from GetTemps, it releases it to the Rendezvous node (for SendDP). Otherwise its behavior is similar to the earlier rendezvous models. The SendDP workload releases its DEQ token to represent the entry call to BufIDP's dequeue. Then waits at the Rendezvous node until it receives both the released DEQ token (at end-accept) and one IDP token.

This model ensures that the synchronization conditions hold for the buffer handling—SendDP must have both an IDP and a DEQ token to execute. However, the model does not strictly represent the execution behavior—the simulated BufIDP dequeue processing may execute before the simulated BufIDP enqueue processing for the IDP. The total processing is the same as the actual system, but the simulated RTS processing may not execute at the same time as the actual system. Additional extensions could be added to represent the exact behavior if necessary. In the Nielsen–Shumate case study, the dequeue processing requirement is negligible (0.00001 s), and modeling the exact processing sequence would not affect the overall results, so this model is sufficient.

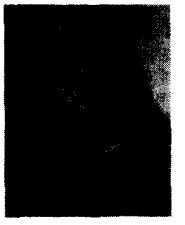
ACKNOWLEDGMENT

We are grateful to Scientific and Engineering Software, Austin, TX for the use of their modeling software in producing the results shown in Tables V–VIII.

REFERENCES

[1] R. A. Ammar and B. Qin, "An approach to derive time costs of sequential computations," *J. Syst. Software,* vol. 11, pp. 173–180, 1990.
[2] G. Arango, "Domain analysis: From art to engineering discipline," in *Proc. 5th Int. Workshop Software Specification and Design,* Pittsburgh, PA, published as *Software Eng. Notes,* vol. 14, pp. 152–159, 1989.
[3] QASE product literature available from Advanced System Technologies, Inc.
[4] T. Baker and G. Scallon, "An architecture for real-time software systems," Tech. Rep. 85-06-04, Dep. Comput. Sci., Univ. of Washington, Seattle, WA, July 1985.
[5] G. Balbo, S. Bruell, and S. Ghanta, "Combining queueing networks and generalized stochastic Petri net models for the analysis of some software blocking phenomena," *IEEE Trans. Software Eng.,* vol. SE-12, no. 4, pp. 561–576, 1986.
[6] M. Baldassari, B. Bruno, V. Russi, and R. Zompi, "PROTOB: A hierarchical object-oriented CASE tool for distributed systems," in *Proc. Europ. Software Eng. Conf.,* 1989, Coventry, England, Sept. 1989.
[7] M. Baldassari and G. Bruno, "An environment for object-oriented conceptual programming based on PROT nets," in *Advances in Petri Nets, Lectures in Computer Science No. 340,* Berlin: Springer-Verlag, 1988, pp. 1–19.
[8] G. Booch, *Software Engineering with Ada,* second ed. Menlo Park, CA: Benjamin/Cummings, 1987.
[9] R. J. A. Buhr, *System Design with Ada.* Englewood Cliffs, NJ: Prentice-Hall, 1984.
[10] R. J. A. Buhr, G. M. Karam, C. J. Hayes, and C. M. Woodside, "Software CAD: A revolutionary approach," *IEEE Trans. Software Eng.,* vol. 15, no. 3, pp. 235–249, 1989.
[11] T. M. Burger and K. W. Nielsen, "An assessment of the overhead associated with tasking facilities and task paradigms in Ada," *Ada Lett.,* vol. VII, no. 1, pp. 49–58, 1987.
[12] Product literature available from Cadre Technologies.
[13] G. W. Cherry, *The PAMELA 2 Designer's Handbook.* Reston, VA: ThoughtTools, Inc., 1987.
[14] G. Chiola, "Great SPN 1.5 software architecture," in *Proc. 5th Int. Conf. Modeling Techniques and Tools for Comput. Perform. Eval.,* Torino, Italy, Feb. 1991, pp. 117–132.
[15] G. Ciardo, J. Muppala, and K. Trivedi, "SPNP: Stochastic Petri net package," in *Proc. Conf. Petri Nets and Performance Models,* Kyoto, Japan, Dec. 1989, pp. 142–151.
[16] D. Craigen, "FM 89: Assessment of formal methods for trustworthy computer systems," in *Proc. 12th Int. Conf. Software Eng.,* Nice, France, 1990, pp. 233–235.
[17] D. W. Embley, B. C. Kurtz, and S. N. Woodfield, *Object-Oriented Systems Analysis: A Model-Driven Approach.* Englewood Cliffs, NJ: Yourdon, 1992.
[18] G. Estrin, R. S. Fenchel, R. R. Razouk, and M. K. Vernon, "SARA (System Architect's Apprentice): Modeling, analysis, and simulation support for design of concurrent systems," *IEEE Trans. Software Eng.,* vol. SE-12, no. 2, pp. 293–311, 1986.
[19] G. Fox, "Performance engineering as a part of the development life cycle for large-scale software systems," in *Proc. 11th Int. Conf. Software Eng.,* Pittsburgh, PA, 1989, pp. 85–94.
[20] A. Gabrielian and M. K. Franklin, "Multi-level specification and verification of real-time software," in *Proc. 12th Int. Conf. Software Eng.,* Nice, France, Mar. 1990, pp. 52–62.
[21] D. Harel, "Statecharts: A visual formalism for complex systems," in *Science of Computer Programming,* vol. 8, D. Gries, Ed., 1987, pp. 231–274.
[22] B. R. Haverkort, I. G. Niemegeers, and P. V. vanZanten, "DyQNtool: A performability modeling tool based on the dynamic queueing network concept," in *Proc. 5th Int. Conf. Modeling Techniques and Tools for Comput. Perform. Eval.,* Torino, Italy, Feb. 1991, pp. 174–188.
[23] N. R. Howes, "Toward a real-time Ada design methodology," in *Proc. TRI-Ada,* 1990, pp. 189–203.
[24] N. R. Howes and A. C. Weaver, "Measurements of Ada overhead in OSI-style communications systems," *IEEE Trans. Software Eng.,* vol. 15, no. 12, pp. 1507–1517, 1989.
[25] M. Jackson, *System Development.* Englewood Cliffs, NJ: Prentice-Hall, 1983.
[26] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, and B. E. Melhart, "Software requirements analysis for real-time process control systems," *IEEE Trans. Software Eng.,* vol. 17, no. 3, pp. 241–258, 1991.
[27] F. Jahanian and A. K. L. Mok, "A graph-theoretic approach for timing analysis and its implementation," *IEEE Trans. Comput.,* vol. C-36, no. 8, pp. 961–975, Aug. 1987.
[28] R. Jain, *Art of Computer Systems Performance Analysis.* New York: Wiley, 1990.
[29] M. Joseph and P. Pandya, "Finding response times in a real-time system," *Comput. J.,* vol. 29, no. 5, pp. 390–395, 1986.
[30] K. Kant, "Modeling interprocess communication in distributed programs," in *Proc. Int. Conf. Petri Nets and Perform. Models,* Aug. 1987, pp. 75–83.
[31] H. Kopetz and W. Merker, "The architecture of Mars," in *Proc. FTCS 15,* Ann Arbor, MI, June 1985, pp. 274–279.
[32] H. Kopetz *et al.*, "The design of real-time systems: From specification to implementation and verification," *Software Eng. J.,* pp. 72–82 1991.
[33] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Englewood Cliffs, NJ: Prentice-Hall, 1984.
[34] E. LeMer, "MEDOC: A methodology for designing and evaluating large-scale real-time systems," in *Proc. NCC, 1982,* Houston, TX, June 1982, pp. 263–272.
[35] N. G. Leveson and J. L. Stolzy, "Safety analysis using Petri nets," *IEEE Trans. Software Eng.,* vol. SE-13, no. 3, pp. 386–397, 1987.
[36] S.-T. Levi and A. K. Agrawala, *Real-Time System Design.* New York: McGraw-Hill, 1990.
[37] M. A. Marsan, G. Balbo, and G. Conte, "A class of generalized stochastic nets for the performance evaluation of multiprocessor systems," *ACM Trans. Comput. Syst.,* vol. 2, no. 2, pp. 93–122, 1984.
[38] J. F. Meyer, "Performability modeling of distributed real-time systems," in *Mathematical Computer Performance and Reliability,* G. Iazeolla, P. I. Courtois, and A. Hordijk, Eds. Amsterdam: Elsevier, 1984.
[39] M. K. Molloy, *Fundamentals of Performance Modeling.* New York: Macmillan, 1989.
[40] D. M. Neuse, J. C. Browne, and P. Jain. "Timing analysis of hierarchical models for real-time systems," in *Proc. Nat. Conf. Methodologies*

and Tools for Real Time Syst., National Insitute for Software and Productivity, Washington, DC, July 1989.
[41] D. M. Neuse and J. C. Browne, "Graphical tools for software system performance engineering," in *Proc. CMG XIV,* Washington, DC, Dec. 1983, pp. 353–355.
[42] K. W. Nielsen and K. Shumate, "Designing large real-time systems with Ada," *Commun. ACM,* vol. 30, no. 8, pp. 695–715, 1987.
[43] PIWG, "Ada performance issues," *Ada Lett.,* vol. X, no. 3, 1990.
[44] R. Prieto-Diaz, "Domain analysis: An introduction," *Software Eng. Notes,* vol. 15, no. 2, pp. 47–54, 1990.
[45] Product literature available from Ready Systems.
[46] J. A. Rolia, "Predicting the performance of software systems," Ph.D. dissertation, Univ. of Toronto, 1992.
[47] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice-Hall, 1991.
[48] B. Sanden, "Entity-life modeling and structured analysis in real-time software design—A comparison," *Commun. ACM,* vol. 32, no. 12, pp. 1458–1466, 1989.
[49] C. H. Sauer and K. M. Chandy, *Computer Systems Performance Modeling.* Englewood Cliffs, NJ: Prentice-Hall, 1981.
[50] L. Sha and J. B. Goodenough, "Real-time scheduling theory and Ada," *IEEE Comput.,* vol. 23, no. 4, pp. 53–62, 1990.
[51] S. Shlaer and S. J. Mellor, *Object-Oriented Systems Analysis.* Englewood Cliffs, NJ: Yourdon Press, 1988.
[52] H. Sholl and S. Kim, "An approach to performance modeling as an aid in structuring real-time distributed systems software," in *Proc. 19th Hawaii Int. Conf. Syst. Sci.,* Honolulu, HI, Jan. 1986, pp. 5–16.
[53] C. U. Smith, *Performance Engineering of Software Systems.* Reading, MA: Addison-Wesley, 1990.
[54] C. U. Smith and L. G. Williams, "Why CASE should extend into software performance," *Software Mag.,* vol. 10, no. 9, pp. 49–65, 1990.
[55] C. U. Smith, "Performance engineering," in *The Encyclopedia of Software Engineering.* New York: Wiley, 1994
[56] R. M. Smith, K. S. Trivedi, and A. V. Ramesh, "Performability analysis: Measures, an algorithm, and a case study," *IEEE Trans. Comput.,* 1988.
[57] J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A new paradigm for real-time systems," *IEEE Software,* vol. 8, no. 3, pp. 62–72, 1991.
[58] L. VanZijl, D. Milton, and S. Crosby, "A tool for graphical network modeling and analysis," *IEEE Software,* vol. 9, no. 1, pp. 47–54, 1992.
[59] P. T. Ward and L. G. Williams, "Domain modeling," Tech. Rep. SERM-012-90, Software Engineering Research, 1990.
[60] ——, "Domain analysis: An example," Tech. Rep. SERM-013-90, Software Engineering Research, 1990.
[61] P. T. Ward, "The transformation schema: An extension of the data flow diagram to represent control and timing," *IEEE Trans. Software Eng.,* vol. SE-12, no. 2, pp. 198–210, 1986.
[62] C. M. Woodside, E. M. Hagos, E. Neron, and R. J. A. Buhr, "The CAEDE performance analysis tool," *Ada Lett.,* vol. XI, no. 3, 1991.
[63] C. M. Woodside, "Throughput calculation for basic stochastic rendezvous networks," *Perform. Eval.,* vol. 9, 1989.
[64] J. Xu and D. L. Parnas, "On satisfying timing constraints in hard real-time systems," in *Proc. ACM SIGSOFT 91 Conf. Software for Critical Syst.,* New Orleans, LA, Dec., published as *Software Eng. Notes,* vol. 16, no. 5, pp. 132–145, 1991.
[65] S. J. Young, *Real Time Languages: Design and Development.* Chichester, England: Ellis Horwood, 1982.



Connie U. Smith (S'79–M'80–SM'87) received the B.A. degree from the University of Colorado, Boulder, and the M.A. and Ph.D. degrees in computer science from the University of Texas at Austin.

She is principal consultant with Performance Engineering Services, Santa Fe, NM, specializing in seminars, consulting, and products to support Software Performance Engineering (SPE). She is the author of *Performance Engineering of Software Systems* (Reading, MA: Addison-Wesley, 1990), and numerous scientific papers. She has over 20 years experience in computing, 16 of which have been in the practice, research, and development of performance prediction techniques. Her research interests include software performance engineering, performance modeling, performance engineering for safety-critical systems, computer graphics, and object-oriented development.

Dr. Smith received the Computer Measurement Group's 1986 AA Michelson Award for technical excellence and professional contributions for her SPE work. She has been an officer of ACM SIGMETRICS (1983–1993), is a past ACM National Lecturer, and is an active member of the Computer Measurement Group.

Lloyd G. Williams received the Ph.D. degree in physical chemistry from the University of Wisconsin.

He is President of Software Engineering Research, a consultation, research, and training firm specializing in preimplementation support for real-time systems development. His interests include object-oriented development, software performance engineering, and software tools and environments.

Dr. Williams is a member of the Association for Computing Machinery and the IEEE Computer Society.