An Integrated Instrumentation Environment for Multiprocessors

ZARY SEGALL, AJAY SINGH, RICHARD T. SNODGRASS, MEMBER, IEEE, ANITA K. JONES, SENIOR MEMBER, IEEE, AND DANIEL P. SIEWIOREK, FELLOW, IEEE

***Abstract*—This paper introduces the concept of an integrated instrumentation environment (IIE) for multiprocessors. The primary objective of such an environment is to assist the user in the process of experimentation. The emphasis in an IIE is on experiment management (including stimulus generation, monitoring, data collection and analysis), rather than on techniques for program development as in conventional programming environments. We believe the functionality of the two environments should eventually be provided in one comprehensive environment.**

An experiment *schema* is introduced as an appropriate structuring concept for experiment management purposes. *Schema instances* capture the results of an experiment for later analysis. An example is developed in some detail to demonstrate the potential benefits of such an approach. The three primary components of the IIE, namely, the *schema manager*, the *stimulus generator*, and the *monitor*, are briefly described. A preliminary implementation of the design on the Cm* multiprocessor is briefly discussed.

***Index Terms*—Automated testing, experimentation, experiment management, instrumentation, monitoring, multiprocessor performance evaluation, programming environment, stimulus generation, workload generation.**

I. INTRODUCTION

MULTIPROCESSOR designs have long been proposed to meet the need for powerful, cost-effective computers. Several multiprocessors have been built to study the various tradeoffs inherent in this approach [1]-[9]. An important objective of experimentation in performance evaluation and reliability is to provide evidence to validate the design decisions of these systems. Due to the increased number of independent components in multiprocessors, the space of possible experiments for such machines is orders of magnitude larger than for conventional uniprocessors. There is, therefore, a need to approach the problem of experimentation on multiprocessors in a structured manner.

Manuscript received February 2, 1982; revised July 22, 1982. This work was supported by the Ballistic Missile Defense Advanced Technological Center under Contract DASG60-81-0077 and in part by a National Science Foundation Fellowship. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the B.M.D.A.T.C. or the U.S. Government.

Z. Segall, A. K. Jones, and D. P. Siewiorek are with the Department of Computer Science, Carnegie-Mellon University, Schenley Park, Pittsburgh, PA 15213.

R. T. Snodgrass was with the Department of Computer Science, Carnegie-Mellon University, Schenley Park, Pittsburgh, PA 15213. He is now with the Department of Computer Science, University of North Carolina, Chapel Hill, NC 27514.

A. Singh was with the Department of Electrical Engineering, Carnegie-Mellon University, Schenley Park, Pittsburgh, PA 15213. He is now with Tandem Computers, 19333 Vallco Parkway, Cupertino, CA 95014.

Instrumentation of the machine is the first important step. Typical instruments discussed in the literature include software, hardware, hybrid, and computer network monitors, natural and synthetic workload generators, data compaction tools, and data analysis packages [10]-[18]. Most systems possess multiple instruments which have been built independently over a period of time with little effort toward integration. This unstructured approach has several disadvantages. First, an experimenter has to communicate with each instrument through its unique user interface, requiring familiarity with several sets of conflicting conventions in syntax and data formats. Second, data from one tool have to be converted manually to the format requirements of any subsequent tools. Furthermore, to make correlations across experiments, the experimenter has to manually keep track of experiment dates, input parameters, monitored results, system configuration, and so forth. Finally, getting the tools to interact during the course of the experiment is usually impossible.

Work in the direction of integrating instruments is found primarily in the area of computer network monitors [16]. Nutt [19] observes that the techniques for gathering measurement data have not been effectively used. Although the raw power of existing tools is quite adequate, the use of these tools is often so complex that experiments cannot fully utilize their functionality.

This paper recognizes the need for better human-engineered environments for experimentation with multiprocessors. It introduces the concept of an *integrated instrumentation environment* (IIE) as a structured approach to facilitate the process of experimentation. The design presented emphasizes the integration of several instrumentation tools, including stimulus generation and monitoring, into a unified experiment management environment. An experiment script (a *schema*) is introduced as an appropriate structuring concept for experiment-management purposes. *Schema instances* are introduced to capture the results of an experiment for later analysis. A preliminary implementation of the design on the Cm* multiprocessor [1] under both the StarOS [20] and Medusa [21] operating systems is briefly discussed.

Although some program management concepts have been borrowed from conventional programming environments (PE's) [22]-[24], the thrust in the IIE is substantially different from what is typically discussed in conjunction with programming environments. The emphasis in an IIE is on experiment management, from stimulus generation to monitoring data collection and analysis, rather than on techniques for program development. We believe the functionality of the two should eventually be provided in one comprehensive environment. The IIE draws on the functions provided by PE's such as program specification and translation, version control, multiple programmer support, and module management. This paper assumes the existence of a PE and will therefore not discuss such functionality in the IIE.

Section I-A presents the functions to be performed by the IIE. The basic components of the design of the IIE are presented in Section II. Stimulus specification and representation is discussed in Section II-A. Section II-B discusses the techniques used to collect and process monitoring information. The run-time environment, presented in Sections II-C and II-D, is a system-specific component permitting remote monitoring and stimulus control. Section II-E discusses the schema manager as the central control component supporting the execution of experiments. A preliminary implementation of the IIE on Cm* is discussed in Section III. Relevant portions of an example are discussed throughout the paper to illustrate some of the concepts

A. Functionality of the IIE

An integrated instrumentation environment (IIE) consists of a set of tools which cooperate closely and present the user with a single uniform interface in order to assist and partially automate the process of experimentation. The general objective of an experiment is to inquire about performance, reliability, or any of a number of interesting properties of a computation. In the context of a computer system an experiment is the execution of an instrumented program in a controlled environment allowing measurement, collection, and analysis. An experiment may involve multiple executions of the instrumented program with different input parameters or within different environments.

The IIE supports the notion of an *experiment schema* as the high level unit of experimentation management. Each schema specifies a related collection of *runs*, that is, executions of an instrumented program. Intuitively, a schema can be seen as a parameterized experiment script, describing the experimentation process. A schema specifies the instrumented program, the monitoring directives, the specifications of the run-time environment, and the input parameters for each run.

The result of an execution of a schema is captured in a *schema instance*, containing measurements, values of schema parameters and environmental information. This is a data structure representing the unit of management for the experiment results. Schema instances are archived in a database for later analysis.

By using the generic notions of schema and schema instance the experimentation process can be expressed as in Fig. 1. Each phase of the experimentation process will be discussed in detail in the following sections.

An IIE requires software to support the several phases of experimentation, including

- translation of collections of user-defined modules and predefined synthetic actions into instrumented parallel programs;

```
Schema = DESIGN(Experiment)
WHILE (Not End of Experiment) DO
 BEGIN
   EXECUTE(Schema)
   CREATE(Schema Instances)
 END
ANALYZE(Schema Instances)
```

Fig. 1. Experimentation process in the IIE.

- creation of the schema by merging the instrumented parallel stimulus, the monitoring directives and the environment information;
- schema interpretation and run-time control;
- creation of schema instances; and
- analysis of schema instances.

In order to further illustrate the experimentation process described above, we will follow an example through in some detail. This example shows how the IIE, at each stage, interacts with the user, performs the required actions, and generates its outputs. The example stimulus, called, simply, "a multiprocessor experiment" or MPX, involves a single *initiator* and multiple *servers* communicating through a shared buffer or mailbox. The initiator repeatedly sends requests through the buffer to one or more servers, which operate on those requests concurrently. When the buffer is empty, the servers wait for further requests; when the buffer is full, the initiator waits for a request to be removed by a server.

The servers perform identical functions, so a request can be satisfied by any server. Additionally, the servers communicate with each other via shared memory. The goals of the proposed experiment are to investigate

- the interaction between the request rate (expressed as the average number of requests per unit time) and the number of servers, and
- the effect of the request rate and the number of servers on the average buffer queue length, and the average waiting time in the buffer.

There are two interesting steady-state behaviors that have different average queue length and service rate. In the first case, the request rate exceeds the aggregate processing rate of the servers, and hence, the buffer will always be full. In the other case, the buffer will always contain at most one request.. The aggregate service rate will be approximately constant, yet radically different, in both cases. This analysis assumes a constant individual service rate by independent servers. However, in Cm*, accessing shared data perturbs the performance of both the servers and the buffer insert/remove operations in nonobvious ways, greatly complicating analytical modeling at the queue length and waiting time. As was shown above, the boundary between the two cases is quite distinct if contention is ignored. The experiment will investigate the boundary in the presence of contention.

To summarize our approach, experiments are described as schemata, and the result of executing a schema is a schema instance. The primary functions of the IIE are the creation of schemata, schema management execution, and control of schemata, along with the creation, management and analysis of schema instances. The next section presents the design of an IIE supporting these functions.

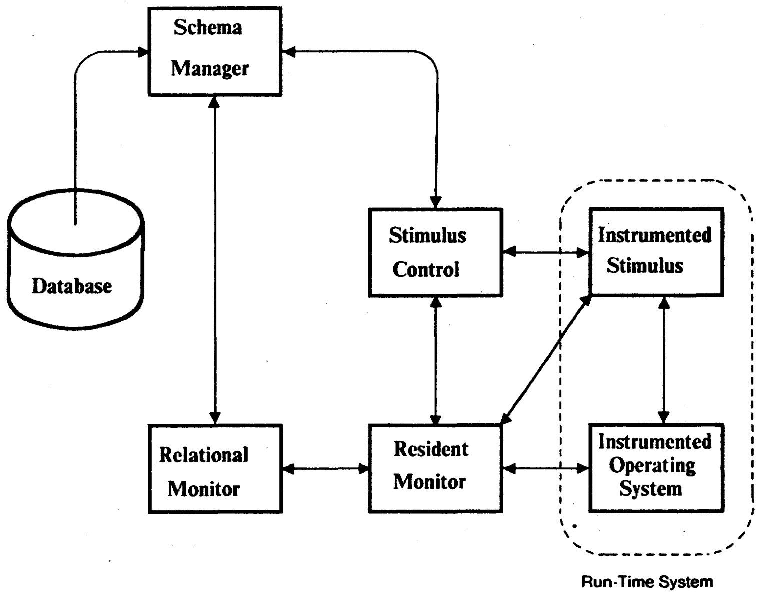


Fig. 2. IIE components. The arcs indicate transfer of data or control.

II. Design of the IIE

The IIE contains several components: a schema manager, a run-time environment, an instrumented stimulus and operating system, a database, and a monitor (see Fig. 2). The monitor consists of a resident monitor, which gathers the data from the system under test, and a relational monitor, which aggregates and correlates the data into a high-level form. The user interacts directly with the schema manager, which communicates with the run-time environment and the monitor, which in turn interacts with the instrumented program (the *stimulus*) and the database. The IIE interacts with the PE through the database.

The schema manager is responsible for supporting the schema and schema-instance abstractions. The monitor initializes the schema instance with information specifying this environment, including details on the hardware configuration, the version of the operating system, support software and stimulus, and the values of the parameters to remain constant for this execution of the schema. The schema manager then cycles through the runs as indicated in the schema, initializing parameters that vary on a per-run basis, starting the stimulus, and collecting the monitoring data. Finally, data concerning the runs as a whole is collected or computed, and stored in the schema instance for later study. Note that not all the IIE components should necessarily reside and execute in the same machine. In fact the Cm* IIE implementation spans several computer systems. The run-time system and the stimulus are resident in Cm*, whereas the schema manager, the database and the relational monitor are remotely located in a VAX 11/780. The two computer systems are connected by an Ethernet link.

One motivation for partitioning the components of the IIE into a run-time environment and a remote environment is that only the run-time environment is constrained to any particular hardware or software configuration. Care has been taken to make the remote components as system independent as possible. Currently, two preliminary implementations exist for the run-time environment for two different operating systems, while only one implementation of the remote components was necessary (see Section III).

The stimulus controller component provides a well-defined interface to the instrumented stimulus. The functions it supports include modifying parameters within the stimulus both before and during the run, generating initial control events for the stimulus, reporting errors back to the schema manager, and controlling the clock. Similarly the resident monitor provides a uniform interface for the relational monitor. The resident monitor is responsible for enabling and disabling *sensors* and for sending the information back to the relational monitor in a format convenient for further processing. The sensors are embedded in the stimulus, in the stimulus controller, in the operating system, and in the resident monitor itself. The relational monitor controls the resident monitor and computes derived information which is then stored in a schema instance in the database.

The database serves an important role in the IIE because the information contained in the database is the end result of the entire experimentation process. Additionally, the interaction between the IIE and the PE occurs via the database by having one environment create objects in the database for the other environment to use. For instance, schemata are initially created in the PE, to be interpreted by the schema manager. Schema instances, created by the IIE, are managed using the version-control facilities of the PE. By using a common database, it is possible to make use of the functionality provided by the PE. This approach allows the designers of an IIE to concentrate on those operations unique to experiment management.

A. *The Instrumented Stimulus: Representation and Specification*

The stimulus is an arbitrary set of processes executing in parallel. The stimulus itself may incorporate sensors; in addition, sensors reside in the operating system and in the resident monitor. We have developed tools to aid in the rapid development of a stimulus. One of them is a workload generator. A user specifies the behavior of his parallel program in a special high-level behavior-description language, the *B-language*. This behavior is specified as a directed data flow graph, similar to a complex bigraph [25], [26]. The nodes of the graph represent subtasks, or processes, that execute in parallel with other subtasks. Each subtask is composed of *actions*, parametrized program fragments that may be predefined or user-defined, repeated at certain rates. Associated with each arc is a buffer which may hold data variables or control tokens flowing from one subtask to another. Each subtask has an associated control tuple (i, o) where i corresponds to the in-firing rule for the subtask and o corresponds to the out-firing rule. This set of firing rules characterizes the precedence relationship between the subtasks of the graph. A B-language program is compiled into an executable version as illustrated in Fig. 3. This section gives a brief overview of the B-language; a more detailed discussion can be found in [27].

The B-language thus represents the interaction of parallel processes via the graph model of computation. A typical example is shown in Fig. 4. Subtask $A1$ is fired by the arrival of a token in buffer $B1$ which corresponds to the entry arc of the graph. Upon completion, subtask $A1$ fires either of subtasks $A2$ or $A3$ by placing control tokens in either buffers $B2$ or $B3$, respectively. There is a certain probability associated with the OR-output logic of subtask $A1$ (designated by the "+"). Finally, subtask $A4$ fires if it receives a token either from $A2$ or $A3$. Upon completion, it places a token in buffer $B6$, which corresponds to the exit arc of the graph and represents the end of a single execution of the parallel synthetic program. The B-language subtask declarations for this example are as follows.

```
SUBTASK A1 {INLOGIC: B1; OUTLOGIC: %40(B2)
                                   OR %60(B3)}
        .
        .
SUBTASK A2 {INLOGIC: B2; OUTLOGIC: B4}
        .
        .
SUBTASK A3 {INLOGIC: B3; OUTLOGIC: B5}
        .
        .
SUBTASK A4 {INLOGIC: B4 OR B5; OUTLOGIC: B6}
```

Notice that the buffers B_1–B_6 correspond to the arcs of the parallel synthetic program. The delimiter "%" is used to specify the branching probabilities for the arcs of an OR output.

The specification of parallel synthetic programs in the B-language is based on the object model supported by both operating systems on Cm* [20], [21]. The objects represented directly in the B-language include the following.

- The *task force* object: The task force abstraction, a collection of processes that cooperate to achieve a single logical task, is represented by a set of *subtasks*.
- The *subtask* object: This is the sequential computation unit that cooperates with similar user-defined objects to compute the overall stipulated multiprocess task.
- The *buffer* object: The buffer object is a conventional queue of messages and is used by the subtasks to communicate with each other.
- The *semaphore* object: Semaphores synchronize requests for shared resources.
- The *file* object: Files represent a sequence of bytes.
- The *shared* data object: Variables specified in the shared data object are globally shared by all the subtasks of the task force. This allows communication of data and control through shared memory.
- The *table* object: Tables implement functions varying with time.

Within a subtask, the basic building block is an action. To capture the cyclic nature of synthetic workloads, an action a_j itself is described by an action-repetition tuple (specified as $\langle a_i, r_i \rangle$). This tuple specifies that the action a_i is repeated sequentially r_i times, constituting action a_j. An action may be arbitrarily complex, and may be further composed of action-repetition tuples. Also, both the a and the r can be parametrized. Other control constructs within a subtask include composition and conditional and probabilistic branching.

The library of actions consists of a collection of predefined and user-defined program fragments, programmed in the systems programming language and stored as part of the system database. Examples of predefined actions include sending or receiving messages via a buffer, inputting or outputting to a file, referencing local memory, blocking on a semaphore, and accessing a shared resource. The user gains flexibility by being able to include his own special program fragment among the actions in the library. An example of a user-programmed action is the code for a disk process in a database application running on a specific multiprocessor. Hence, the library of actions is specific for a particular multiprocessor system. The B-language should be viewed as a portable framework into which system specific actions are inserted from a library of actions.

Special control constructs are included in the B-language so that the schema manager may control the user's workload at run-time as specified in the schema. The control commands initiated by the schema manager are executed by the stimulus controller component of the run-time system. The VARY construct in the language permits the stimulus controller to vary parameters on a per-run basis. The language also allows one to specify that the parameters are to vary in real time. This is accomplished by binding a real-time function to a run-time variable on a per-run basis. The real-time function is defined by a table object and an associated interval of time. The stimulus controller forces the run-time variable to take on successive values from the table during successive time intervals.

Using the MSGEVENT construct, the language permits the stimulus controller to initiate variable time-driven events in the stimulus on a per-run basis. This construct requests the stimulus controller to deliver messages to a buffer with inter-message time periods as specified by successive entries of a table. The stimulus controller can associate a different table object, or a constant time-period, with the MSGEVENT variable on a per-run basis.

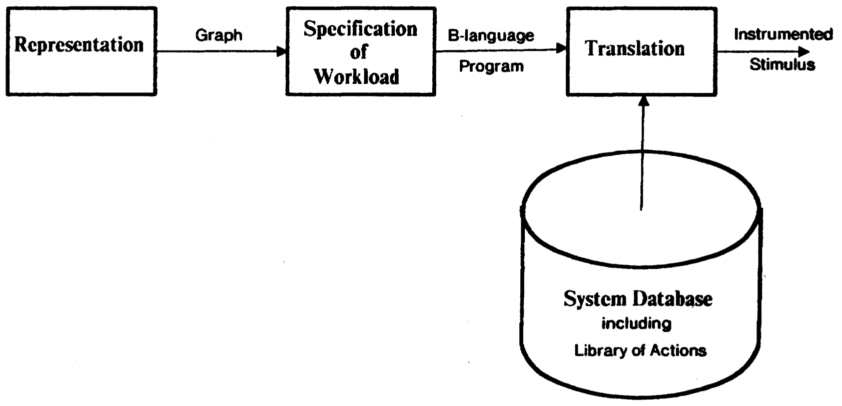

Representation
Graph
Specification of Workload
B-language Program
Translation
Instrumented Stimulus
System Database including Library of Actions

Fig. 3. Stimulus generation steps.

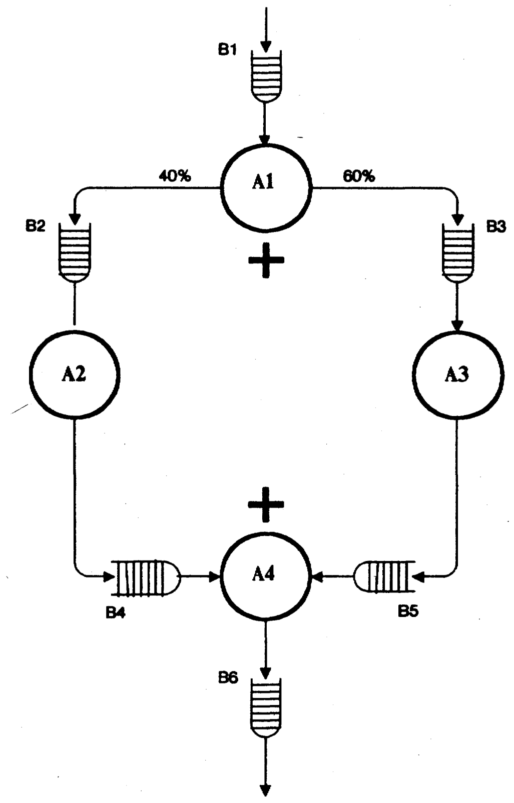


Fig. 4. A parallel synthetic program—graph representation.

```
TASKFORCE MPXperiment ;
BUFFER
   RequestBuffer{ SIZE: 512 } ;
SEMAPHORE
   GDSemaphore{ INITIAL: 1 } ;
SHARED
   GlobalData[512] ;
VARY
   RequestPeriod;
MSGEVENT
   RequestService = RequestBuffer @ RequestPeriod ;
SENSOR
   StartGlobalPhase ;

SUBTASK Servers[1..5]
      { INLOGIC : RequestBuffer }
VARY
   SharedDataAccess ;
BEGIN
   <$DoLocalWork : 10>,
   StartGlobalPhase,
   <$AccessSharedData(GDSemaphore,GlobalData): SharedDataAccess>
END
```

Fig. 5. B-language program for the MPX.

To allow measurement of the generated workload, a special SENSOR construct permits a user to embed sensors into his program. Sensors allow specified information as well as a timestamp to be sent to the monitor as event records. In addition to user-defined sensors, the B-language program has some built-in sensors. For example, the start time and end time for each execution of a subtask are automatically recorded in the event record. Furthermore, instrumentation available in the operating system and the IIE run-time system allows the schema manager to access information not explicitly specified in the B-language program. An example is information regarding the interaction of the stimulus and the operating system.

The B-language translator constructs special data structures allowing the stimulus controller to exercise external control over the experiment as specified in the B-language program. The translator also generates sensor descriptions (see Section II-D) for all programmed and predefined sensors in the B-language program. These descriptions are used by the relational monitor to sort out event records flowing from the resident monitor.

As an example, consider the B-language program (Fig. 5) for the single-requester, multiple-server experiment discussed in Section II. The task force consists of an array of five identical server subtasks that wait on the RequestBuffer for queued service requests. The RequestBuffer is associated with the message-event generator via the MSGEVENT construct. This allows an experimenter to vary the request rate by changing the time (RequestPeriod) between successive firing of servers on a per-run basis. The BEGIN and END constructs mark the service loop of each subtask which is executed each time its in-firing rule is satisfied. In this example, each server does ten units worth of work local to its processor, and then does some variable number of accesses to global data, which is arbitrated by a semaphore. A sensor, StartGlobalPhase, is embedded in each subtask and sends an event record to demarcate the transition from local work to global work. Built-in sensors record the begin and end of each subtask and the firing of request tokens by the message-event generator. The variable parameters of the experiment are the number of active servers, the request rate, and the amount of global work done by each server. This program will be specified in the schema as the stimulus for the MPX.

B. Relational Monitor

In the IIE, each time the experiment schema is interpreted, and the stimulus executed one or more times, various monitoring information is collected and stored in the database in a schema instance.

The model of the monitoring data adopted in the IIE is a variant of the *relational model* used in conventional relational databases [28]. Information is recorded as a collection of two-dimensional tables, called *relations*. Each row, called a *tuple*, records a particular relationship between entities named in the columns, called *domains*, of the tuple. For example, the relation *Running* (*Process*, *Processor*), with two domains, may contain the tuple (*MyProcess*, *ProcessorA*), indicating that the process called *MyProcess* is running on the processor called *ProcessorA*. Relations used in monitoring are temporal in that each tuple records relationships that are true at an instance of time or over some interval of time. A relation involving instances of time is called an *event relation*; each tuple records the occurrence of a particular event. A *period relation*, on the other hand, records a relationship that exists for an interval of time. Periods are delimited by events; each tuple (period) in the *Running* relation is associated whith two other event tuples, one in the *Start* relation and one in the *Stop* relation. Time is included in an implicit domain manipulated by the monitor.

The *Running* relation is an example of a *primitive relation* because the information contained in the relation is a direct translation of a set of recorded events. Primitive relations may be divided into three categories: operating system, stimulus control, and user-defined. The first category is concerned with information involving the operations and data structures supported by the operating system. The *Running* relation is in this category. The second category involves the actions performed by the run-time portion of the IIE. Examples of event relations from the MPX include

- *RequestService*(*TokenID*): the sending of a MsgEvent token to RequestBuffer; the TokenID identifies the token;
- *ServersStart*(*Index*, *TokenID*): the in-firing of a sensor's subtask; the *Index* identifies the Server; the *TokenID* identifies the token causing the firing;
- *ServersEnd*(*Index*, *TokenID*): the out-firing of a sensor's subtask.

The one user-defined primitive relation specified in the MPX, *StartGlobalPhase*, is also an event relation and contains only the implicit time domain. This relation was declared as a sensor in the B-language program for the MPX (see Fig. 5), and records the time at which the Server subtask finished its local work and started the shared data access.

Given a collection of primitive relations, new relations can be defined as a result of operations performed on existing relations. These *derived relations* are specified using a relational query language. The query language used in the IIE is a version of Quel [29] augmented with additional temporal constructs and is discussed elsewhere [30]. Fig. 6 illustrates the definition of the derived relations AverageQLength and ServiceRate used in the MPX. The former relation has one domain, AvQL, with the tuples specifying this value for the various time intervals. Similarly, the ServiceRate relation will have one domain, SRate, containing values varying over time. These queries will be referred to by the schema for the MPX, and will specify both the primitive relations to be monitored and the calculations to be performed on the data in the event records.

C. Stimulus Controller

The stimulus controller component of the run-time system is a set of utilities that permit control of the stimulus as specified in the schema. While the schema manager provides experiment *management* through the management of the schema abstraction, the stimulus controller provides low-level experiment *control* through the management of a single run. The motivation was to separate the low-level control functions from the experiment-management functions so that different management strategies could be carried out using common control primitives. The functions exported by the stimulus controller are therefore geared towards the initialization and execution of a single run.

One responsibility of stimulus controller is to ensure the repeatable behavior of a run by eliminating side effects from one run that might perturb the next run. An example of a side effect is the presence of tokens left over in the edges (buffers) as a result of the previous run. The stimulus controller ensures that all data structures are in a well defined state at the beginning of a run. For example, buffers are emptied and all semaphores are initialized as specified in the B-language program.

The stimulus controller is also responsible for the variation of parameters on a per-run basis and in real time during a run. The variation of parameters on a per-run basis involves the VARY parameters of the B-language program (see Section II-A), and the variation of the graph-structure representation of the program. A typical modification of the graph structure involves changing the number of active subtasks for a particular run. This is particularly useful in real-time experimentation, where one wants to determine the number of subtasks necessary to meet real-time constraints. The variation of parameters in real time during a run involves the variation of the run-time variables of the B-language program according to some function of time expressed as a table object and an associated interval of time.

The stimulus controller must provide a well defined mechanism to start the run. In the graphical representation of the program this corresponds to firing the entry node, that is, placing a token on the entry arc of the graph. To start a run, the stimulus controller delivers a specified number of control tokens into a system-defined buffer, called the IgnitionBuffer, which corresponds to the entry arc of the data-flow graph. The user may now use this source of tokens to start any desired subtask, by specifying the IgnitionBuffer appropriately in the in-firing rule of that subtask. Similarly, to detect the end of a run, the stimulus component watches a system-defined TerminationBuffer for a specified number of tokens.

The stimulus controller has four major subcomponents. The first subcomponent executes basic control functions, including *initialize*, to initialize the instrumented program before each

```
range of R is RequestService     ; references to R will indicate the
                                 ; RequestService relation
range of S is StartServers
range of Sp is StopServers
define WaitingInQueue (R.TokenID)       ; one domain, the request's
                                        ; TokenID
       where R.TokenID = S.TokenID      ; the request is being serviced
                                        ; by a server
       start R                   ; the waiting begins when the request
       stop S                    ; is made, and ends when the server
                                 ; starts
range of W is WaitingInQueue
define QLength (L = Count(W))    ; count the number of outstanding
                                 ; requests in the buffer
range of Q is QLength
define AverageQLength (AvQL = Average(Q)) ; instantaneous average

define TotalWaiting (W.TokenID)
       where Sp.TokenID = W.TokenID
       start W                   ; total waiting time begins when the
       stop SP                   ; request was made, and ends when the
                                 ; server stops
range of TW is TotalWaiting
define ServiceRate (SRate = 1 / Average(Duration(TW)))
```

Fig. 6. Queries for the MPX.

run; *fire*, to fire a specified number of tokens into the IgnitionBuffer; *vary*, to permit the variation of VARY-parameters on a per-run basis; *display*, to display the value of a *vary*-parameter; *enable/disable*, to enable or disable subtasks on a per-run basis; and *status*, to return the status of the program. Observe that functions such as *display* and *status* are interactive in nature and can be used during the interactive creation of a schema (see Section II-D).

With the second, a message-event generator delivers token messages to prespecified buffers according to prespecified functions of time. Control functions performed by this subcomponent include *start generator*, to start the message-event generator for a particular run; *stop generator;* and *set message event*, to allow the association of either a table object or a constant with a buffer.

With the third, a run-time variable driver ensures that all run-time variables vary in real time as specified by its associated table object and time interval. The main control function of this module is to allow the association of different table objects and time intervals with a run-time variable on a per-run basis.

Finally, with the fourth, a clock module permits access to a set of clocks distributed over the system. This module is used by the message-event generator, the sensors, and the run-time variable driver.

Additional functionality in the instrumented program may be added by augmenting the stimulus controller. For example, a set of components used for experimentation related to reliability has been designed and partially implemented. This includes software-implemented voters, and accelerated fault-insertion and configuration-control modules.

D. The Resident Monitor

The monitoring information is collected as *event records*, generated by *sensors* in the instrumented stimulus, the run-time system, the operating system, or the hardware. Each event record contains an indication of the operation being monitored, the name of the component performing the operation, and the name of the object the operation is being performed on. The event record may optionally contain a timestamp and other information germane to the event. For instance, a sensor located in a file-system process might generate event records for file reads. In this case, the event record would include the name of this process, the name of the file being read, an indication that this is a file-read event, the timestamp, and perhaps the block number being read.

Highly selective filtering of the event records is necessary to constrain the flow of event records into the monitor. Enabling and filtering directives are encapsulated in data structures called *receptacles*, associated with either active components, such as a file-system process, or passive objects, such as a file. Receptacles contain event-enable switches as well as a buffer for temporarily storing event records. The resident monitor (and thus, indirectly, the relational monitor) has the ability to enable switches in each receptacle. The flexibility in associating receptacles with either processes or objects provides a mechanism for filtering the event records. For example, if the receptacle was associated with the file, and the file-read event was enabled, event records for all file reads performed on the file would be written into the receptacle. On the other hand, if the receptacle was associated with a file-system process, event records for all file reads performed by the process on any file would be written into the receptacle.

A task force is instrumented by specifying the sensors, events, and object types in a file, called a *sensor description.* The operating system and stimulus controller, being task forces themselves, are also associated with sensor descriptions. A sensor description is generated automatically when a B-language program is processed. Users may also write their own sensor descriptions if they so desire. Fig. 7 illustrates the sensor description generated from the B-language program for the MPX given in Fig. 5. This description includes a sensor-process definition for each subtask and for the stimulus controller, and events for the start and end of execution of each subtask and the start of each run. Another program takes the sensor description and produces optimized code for each software-implemented sensor, based on the specifications in the sensor description. Sensor descriptions thus allow users to specify their own sensors which will utilize the same mechanisms for event record and generation as the sensors embedded in the run-time and operating systems.

It is important to note that the user never needs to be concerned about receptacles or event records. Instead, the IIE (through the monitor component) presents to the user the view

```
(Taskforce (name MPExperiment)                  ; Standard prelude
   ... )
(SensorProcess (Name StimulusControl)
   ... )
(Event (Name PerRun)
       (Domains (Domain (Name RunNumber)
                        (Type Integer))
                (Domain (Name RequestPeriod)
                        (Type Integer))
                (Domain (Name ServerCount)
                        (Type Integer)))
       (Timestamp yes)
   ... )
(Event (Name RequestService)                    ; MsgEvents
       (Location StimulusControl)
       (Domains (Domain (Name TokenID)
                        (Type Integer)))
       (Timestamp yes)
   ... )
(SensorProcess (Name Servers)                   ; SubTasks
   ... )
(Event (Name ServersStart)
       (Location Servers)
       (Domains (Domain (Name Index)
                        (Type Integer))
                (Domain (Name TokenID)
                        (Type Integer)))
       (Timestamp yes)
   ... )
(Event (Name ServersEnd)
   ... )
(Event (Name StartGlobalPhase)                  ; User-defined sensors
   ... )
...
```

Fig. 7. Sensor description for the MPX.

```
SCHEMA (<invocation parameters>)
        <system configuration>
        <stimulus>
        <monitoring directives>
        <initial conditions>
        <experiment directives>
END SCHEMA
```

Fig. 8. High level organization of a schema.

```
SCHEMA MPX (RequestPeriod, SDA)

SYSTEMCONFIGURATION <configuration data>;

TASKFORCE <B-language program>;

MONITORQUERIES <relational queries>;

RESULTRELATIONS AverageQLength, ServiceRate;

VARY SharedDataAccess[I] = SDA WHERE I FROM 1 TO 5;

VARY NoOfServers FROM 1 TO 5
DO
   BEGINEXPERIMENT
   ENABLE Server[I] WHERE I FROM 1 TO NoOfServers;
   TERMINATE AFTER 30 seconds
   ENDEXPERIMENT
OD
ENDSCHEMA
```

Fig. 9. The schema for the MPX.

of a database composed of temporal relations. New relations can be derived using the query language (identified in Section II-B). As a result of executing a query, the appropriate operations (locating and enabling receptacles, processing event records, and generating the schema instances) are performed automatically.

The use of receptacles and sensors may extend from sensors implemented in hardware to sensors embedded in the operating system to sensors placed in the user's program. It is the resident monitor's responsibility to extract the event records from the receptacle and send them to the relational monitor. By the time the relational monitor receives the event records, they are in an identical format regardless of how they were generated.

E. Schema Management

The central management and control of the schema and the schema instances is performed by the schema manager. Functions of the schema manager fall into two broad categories: the creation, manipulation, and execution of the schema and the creation, archiving, and cross analysis of schema instances. The schema manager is organized in three main functional parts, as follows.

1) A user interface provides a uniform view of the various components of the IIE. Schemata can be created using conventional text editors, or incrementally, by directing the IIE to perform a series of runs. In the latter case, the corresponding schema and schema instance are automatically generated and archived. This incremental mode is particularly helpful in the tuning of experiments. The user interface also directly supports monitoring queries and database queries thereby allowing a user to manipulate and analyze schema instances.

2) A schema interpreter scans the schema and sends control directives to the run-time system, including global initialization commands for the entire experiment along with commands to set up, start, and terminate each run.

3) A schema-instance generator interacts with the relational monitor to ensure that an instance is created and placed in the database. Both predefined and user-defined relations are created and stored in the schema instance as a result of interpreting the schema.

The schema contains all the necessary information to perform a complete experiment. It consists of five major components: the system configuration, the stimulus, monitoring directives, initial experiment conditions, and experiment directives (see Fig. 8). The system configuration completely defines the environment the experiment is to be performed in. The stimulus is in the form of a translated B-language program containing controlling parameters and data-collection sensors as described in Section II-A. The monitoring directives are in the form of a collection of queries as described in Section II-B. The initial experiment conditions consist of a set of invocation parameters and the required resources (i.e., hardware and operating system configuration and instrumentation, stimulus version, etc.). Invocation parameters can be used to initialize parameter values for experiments and are typically specified at schema interpretation time. The experiment directives are interpreted by the schema manager and specify how the stimulus should be executed. Specifications are provided for the iteration of the stimulus over the experiment runs along with the variation of parameters for each run.

During schema execution, the relational monitor creates a schema instance to hold the results of the experiment. The monitor collects all the resulting event records together with the schema identification and environment information and creates an object to be managed by the PE. By using standard relational database queries, the user can then perform analyses across schema instances. The data in the instance which are collected automatically provide the user with enough information to replicate any particular execution of the schema to verify the results.

In order to illustrate the use of schema and schema instance, consider the schema describing the MPX, shown in Fig. 9. The schema has two invocation parameters, RequestPeriod and SDA. The configuration data specify the resources requested

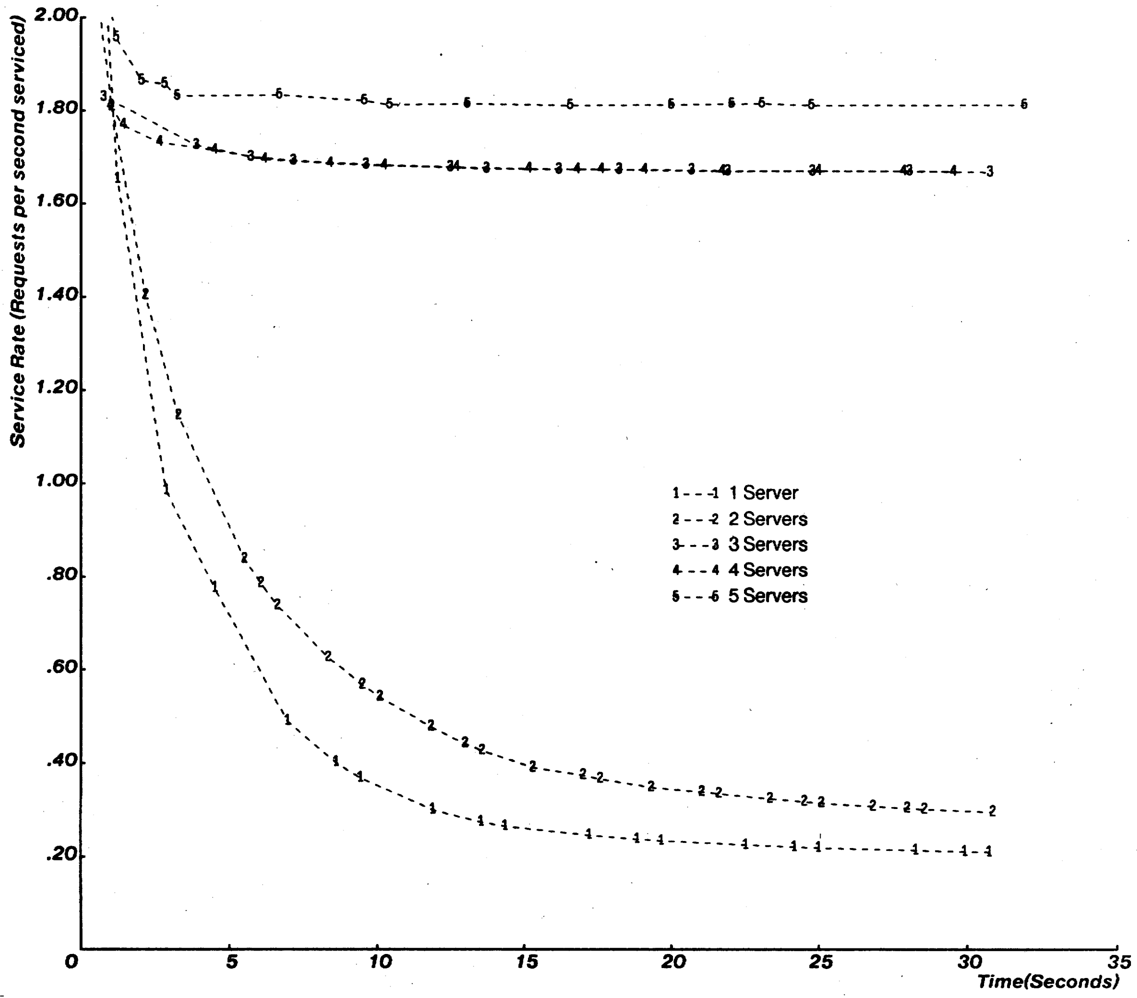


Fig. 10. Service rate versus time for a variable number of servers.

by this experiment, including the versions of the operating system and IIE components, the hardware components, data files to be read by the stimulus, and initial tests to be used later to calibrate the results.

The experiment directives are in the form of a loop which generates the execution of five runs. Each run will have its own value for the NoOfServers parameter. The execution of this schema will terminate when 30 s have passed for each run. During execution, the sensors implanted in the B-language program will generate data which are collected according to the monitor queries.

Each time this schema is interpreted, a schema instance will be automatically created in the database by the IIE. Each instance will have the following components

- the date, time, and user identification;
- the values of the invocation parameters;
- exact version numbers of all software used in the experiment;
- a detailed description of the hardware configuration;
- results of the initial tests as specified in the system configuration; and
- the system- and user-defined relations (in this case, the PerRun, AverageQLength, and ServiceRate relations).

Once the instances have been created, additional analysis can be performed on the instances individually or as a group. Fig. 10 shows the relationship between average service rate and time for a RequestPeriod of 200 ms and a value of Shared DataAccess of 400 accesses per request. Initially the service rate is high, since the buffer is empty. For five servers, the buffer never contains many requests, so the average service rate remains high. However, for less than three servers, the buffer fills up quickly, causing the average service rate to plummet. The behavior with three or four servers is more involved, and further analysis is necessary using different values for the request period and the SDA.

III. Implementation

A. Background

Our research vehicle is the Cm* multiprocessor. Cm* is a 50 processor multiprocessor developed and implemented at Carnegie-Mellon University. Two operating systems, Medusa and StarOS, have been developed for Cm*. In addition, substantial utility software built for Cm* runs on other general-purpose computers.

B. Status

An initial version of the IIE has been partially implemented for Cm*. Two versions of the run-time system have been developed, one for each operating system [30], [27]. A substantial library of actions has accumulated for both operating systems, and work is proceeding on implementing the B-language translator. An initial version of the relational monitor has been developed, including the sensor-description of the processor, although substantial effort is still needed before general queries and multiple schema-instance analysis can be executed [30]. The schema manager is in the final design stages. It is expected that a full implementation of the IIE will be completed by the end of 1982.

IV. Conclusion

The IIE constitutes a systematic approach to the task of experimentation on multiprocessors. This approach emphasizes the integration of the tools used for such experimentation and the development of techniques for experiment management. The tools incorporated into the initial design of the IIE have been oriented primarily toward performance measurement. Work is proceeding in the area of reliability experimentation, specifically to enhance the monitor so that it can function across system failures and to implement fault insertion into the stimulus in a controlled fashion. Another interesting use of the IIE is in automated testing of revised modules in the framework of version control. Future research areas include the integration of the IIE with a multiprocessor PE, the incorporation of hardware monitors and other tools into the IIE, and the development of an IIE supporting experimentation of real-time systems.

Acknowledgment

The authors would like to acknowledge the contributions of some of the concepts, and of the implementation by the other members of the Multiprocessor Performance Evaluation Group: X. Castillo, R. Chansler, I. Durham, P. Highnam, E. Gehringer, and P. Sindhu.

References

[1] A. K. Jones and E. F. Gehringer, Eds., "The Cm* multiprocessor project: A research review," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., July 1980.

[2] R. J. Swan, "The switching structure and addressing architecture of an extensible multiprocessor, Cm*," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, Aug. 1978.

[3] S. H. Fuller and S. P. Harbison, "The C.mmp multiprocessor," Dep. Comp. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-78-148, 1978.

[4] M. Satyanarayanan, *Multiprocessors: A Comparative Study*. Englewood Cliffs, NJ: Prentice-Hall, 1980.

[5] S. A. Ward, "The MuNet: A multiprocessor message-passing system architecture," in *Proc. 7th Texas Conf. Computing Systems*, Nov. 1978, pp. 7-21.

[6] D. Siewiorek, M. Canepa, and S. Clark, "C.mmp: The architecture of a fault-tolerant multiprocessor," in *Proc. 7th Annu. Int. Conf. Fault-Tolerant Computing*, June 1977, pp. 37-43.

[7] R. G. Arnold and E. W. Page, "A hierarchical restructurable multi-microprocessor architecture," in *Proc. 3rd Annu. Symp. Comput. Arch.*, Jan. 1976, pp. 44-45.

[8] T. B. Smith and A. L. Hopkins, "Architectural description of a fault-tolerant multiprocessor engineering prototype," in *Dig. Papers, 8th Annu. Conf. Fault-Tolerant Computing*, June 1978.

[9] S. H. Fuller, J. K. Ousterhout, L. Raskin, P. Rubinfeld, P. S. Sindhu, and R. J. Swan, "Multi-microprocessors: An overview and working example," *Proc. IEEE*, vol. 66, Feb. 1978.

[10] I. Gertner, "Performance evaluation of communicating processes," Ph.D. dissertation, Univ. Rochester, NY, 1980.

[11] H. C. Lucas, Jr., "Performance evaluation and monitoring," *Comput. Surveys*, vol. 3, pp. 79-91, Sept. 1971.

[12] W. Bucholz, "A synthetic job for measuring system performance," *J. IBM Syst.*, pp. 309-318, 1969.

[13] P. F. McGehearty, "Performance evaluation of multiprocessors under interactive workloads," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, 1980.

[14] K. Sreenivasan and A. J. Kleinman, "On the construction of a representative synthetic workload," *Commun. Ass. Comput. Mach.*, 1974.

[15] D. Ferrari, "Workload characterization and selection in computer performance measurement," *IEEE Computer*, pp. 18-24, July-Aug., 1972.

[16] D. E. Morgan, W. Banks, D. P. Goodspeed, and R. Kolanko, "A computer network monitoring system," *IEEE Trans. Software Eng.*, vol. SE-1, Sept. 1975.

[17] L. Raskin, "Performance evaluation of multiple processor systems," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, Aug. 1978.

[18] H. Kobayashi, *Modelling and Analysis: An Introduction to System Performance Evaluation Methodology*. Reading, MA: Addison-Wesley, 1978.

[19] G. J. Nutt, "A survey of remote monitors," NBS Special Publ. 500-42, Jan. 1979.

[20] A. K. Jones, R. J. Chansler, Jr., I. Durham, K. Schwans, and S. R. Vegdahl, "StarOS: A multiprocess operating system for the support of task forces," in *Proc. 7th Symp. O.S. Principles*, Sept. 1979, pp. 117-127.

[21] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "Medusa: An experiment in distributed operating system structure," *Commun. Ass. Comput. Mach.*, vol. 23, pp. 92-105, Feb. 1980.

[22] A. N. Habermann, "An overview of the Gandalf project," Carnegie-Mellon Univ., Comput. Sci., Res. Rep. 1978-1979, 1980.

[23] T. Teitelbaum, "The Cornell program synthesizer: A syntax directed programming environment," Dep. Comput. Sci., Cornell Univ., Ithaca, NY, Tech. Rep. TR80-421, May 1980.

[24] R. Medina-Mora and P. H. Feiler, "An incremental programming environment," *IEEE Trans. Software Eng.*, Sept. 1981.

[25] V. Cerf, "Multiprocessors, semaphores and a graph model of computation," Dep. Comput. Sci., Univ. California, Los Angeles, Tech. Rep. 7223, Apr. 1972.

[26] K. P. Gostelow, "Flow of control, resource allocation, and the proper termination of programs," Ph.D. dissertation, Univ. California, Los Angeles, Dec. 1971.

[27] A. Singh, "Pegasus: A workload generator for multiprocessors," Master's thesis, Dep. Elec. Eng., Carnegie-Mellon Univ., Pittsburgh, PA, 1981.

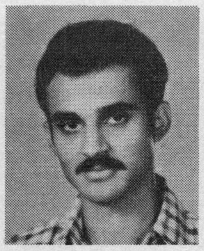
[28] J. D. Ullman, *Principles of Database Systems*. Potomac, MD: Computer Science Press, 1980.

[29] M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of INGRES," *ACM TODS*, vol. 1, pp. 189-222, Sept. 1976.

[30] R. Snodgrass, "Monitoring distributed systems," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, to be published, 1982.

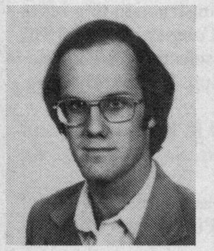
Zary Segall received the E.E. degree from Politechnical Institute, Bucharest, Romania, and the M.Sc. and Ph.D. degrees in computer science from Technion, Haifa, Israel.

Presently, he is on the faculty of the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, and is the head of the Cm* project. His research interests are in the areas of parallel computation, fault tolerance, performance evaluation, and design automation.



Ajay Singh received the B. Tech. degree in electrical engineering from the Indian Institute of Technology, Kankur, in 1980, and the M.S. degree in computer engineering from Carnegie-Mellon University, Pittsburgh, PA, in 1981.

He is currently working with the Operating Systems Group at Tandem Computers Inc., Cupertino, CA. His professional interests include distributed operating systems and performance evaluation.



Richard T. Snodgrass (S'79–M'81) received the B.A. degree from Carleton College, Northfield, MN, in physics, and the M.S. degree from Carnegie-Mellon University, Pittsburgh, PA, in computer science. He is currently completing his studies for the Ph.D. degree at Carnegie-Mellon University, in the area of monitoring distributed systems.

At present he is an Assistant professor at the University of North Carolina, Chapel Hill. His interests include programming environments, databases, expert systems, and user interfaces.

Anita K. Jones (SM'82) received the Ph.D. degree from Carnegie-Mellon University, Pittsburgh, PA, in 1973.

She is presently an Associate Professor with the Department of Computer Science, Carnegie-Mellon University, and a Senior Scientist with Tartan Laboratories Inc., Pittsburgh, PA. She has been a member of program committees for several recent conferences, as well as serving as the Operating Systems Editor for the *Communications of the Association for Computing Machinery*. She is also the Editor-in-Chief of the *ACM Transactions on Computer Systems*, which will commence publication in 1983. Her professional interests include operating systems, distributed systems, protection, parallel algorithms, and the manufacture of software.

Dr. Jones is a member of the Association for Computing Machinery, Sigma Xi, and the IEEE Computer Society.

Daniel P. Siewiorek (S'67–M'72–SM'79–F'81) was born in Cleveland, OH, on June 2, 1946. He received the B.S. degree in electrical engineering (summa cum laude) from the University of Michigan, Ann Arbor, in 1968, and the M.S. and Ph.D. degrees in electrical engineering (minor in computer science) from Stanford University, Stanford, CA, in 1969 and 1972, respectively.

He joined the Departments of Computer Science and Electrical Engineering, Carnegie-Mellon University in 1972, where he is presently Professor of Computer Science and Electrical Engineering. While at Carnegie-Mellon University, he helped to initiate and guide the Cm* project that culminated in an operational 50 processor multi-microprocessor system. He also participated in the Army/Navy Military Computer Family (MCF) project to select a standard instruction set. The ISP (Instruction Set Processor) hardware description language was extensively developed to support the MCF program. He also directed the construction of C.vmp, a triply redundant fault tolerant computer, and formulated a hardware design automation project. His current research interests include computer architecture, reliability modeling, fault-tolerant computing, modular design, and design automation. He has served as Associate Editor of the Computer Systems Department of the *Communications of the Association for Computing Machinery* and is currently serving as Chairman of the IEEE Technical Committee on Fault Tolerant Computing.

Dr. Siewiorek was recognized with an Honorable Mention Award as Outstanding Young Electrical Engineer for 1977, given by Eta Kappa Nu (National Electrical Engineering Honorary Society). He was elected a Fellow of the IEEE in 1981 for "contributions to the design of modular computing systems." He is a member of the Association for Computing Machinery, Tau Beta Pi, Eta Kappa Nu, and Sigma Xi.