# SWSL: A Synthetic Workload Specification Language for Real-Time Systems

Daniel L. Kiskis and Kang G. Shin, *Fellow, IEEE*

***Abstract*— In this paper, we discuss the issues that must be addressed in the specification and generation of synthetic workloads for distributed real-time systems. We describe a synthetic workload specification language (SWSL) that defines a workload in a form that can be compiled by a synthetic workload generator (SWG) to produce an executable synthetic workload. The synthetic workload is then downloaded to the target machine and executed while performance and dependability measurements are made. SWSL defines the workload at the task level using a data flow graph, and at the operation level using control constructs and synthetic operations taken from a library. It is intended to be easy to use, flexible, and capable of creating synthetic workloads that are representative of real-time workloads. It provides a compact, parameterized notation. It supports automatic replication of objects to facilitate the specification of large workloads for distributed real-time systems. It also provides extensive support for the experimentation process.**

***Index Terms*—Synthetic workloads, real-time workloads, distributed real-time systems, performance and dependability measurement experiments**

## I. Introduction

A SYNTHETIC WORKLOAD (SW) is a set of artificial or synthetic programs that execute on a computer system and produce resource demands on the system. The synthetic programs are parameterized to allow the user to easily modify their execution and resource consumption behavior. SW's have long been recognized as useful tools to be used during the experimental evaluation of computer systems. The tasks, or jobs, that make up the SW are selected to represent a particular application domain for a certain class of computers. Early SW's were designed to represent typical business applications running on a mainframe computer, e.g., [4], [17], [26], [36]. SW design was later studied extensively by Ferrari [8]–[10]. However, his work also concentrated on general purpose uniprocessor computers.

At the Real-Time Computing Laboratory of The University of Michigan, Ann Arbor, USA, we are designing and building the Hexagonal Architecture for Real-Time Systems (HARTS) [28]. HARTS is a distributed real-time system consisting of a number of multiprocessor nodes connected by a custom hexagonal mesh interconnection network. HARTS is an experimental system that is to serve as a testbed for developing and evaluating real-time communication, fault tolerance, and operating system concepts. It was decided that an SW would be a valuable tool for this evaluation process. An SW would allow us to create a range of different workloads, each designed to exercise the specific components under study.

The SW's that have been developed for general purpose systems are inadequate for use on a distributed real-time system like HARTS. They are unable to accurately reproduce the behaviors that are characteristic of real-time workloads. Previous SW's developed for real-time systems, e.g., one for FTMP [7] and an early one for HARTS [37], were too inflexible and had no provisions to exercise communication facilities of a distributed system. An SW for a general purpose distributed computer system was proposed by Singh and Segall [29], [30]. Their system, called Pegasus, was to produce SW's for a distributed system. They defined a novel language, the B-language, to describe the SW's. However, no compiler for the B-language was implemented [29], and thus the feasibility and usefulness of the language for specifying SW's for large systems has not been demonstrated. Furthermore, the B-language contains no support for specifying SW's for real-time systems.

Because there are no other sufficiently powerful SW's for distributed real-time systems, we have designed and implemented our own SW [13], [14]. The SW operates in a distributed manner. Each processor executes a set of synthetic tasks and a driver process. The synthetic tasks produce the resource demand for the SW. The driver process initializes and activates the SW and generates stochastic events to simulate random inputs to the workload. It also coordinates with the drivers on the other processors to provide synchronized, distributed control of the SW on the entire multiprocessor. To be useful during experimentation of HARTS, the SW requires a support tool, namely, a synthetic workload generator (SWG). The SWG is needed because of the size of HARTS. Each node on HARTS can have up to three processors for executing application software. Our initial version of HARTS consists of 19 nodes. Hence, an SW for HARTS will consist of programs for 57 processors. Since coding and debugging this number of programs would be a tedious and error-prone undertaking,

Manuscript received January 4, 1992; revised April 8, 1993. This work was supported in part by the National Science Foundation (NSF) under Grants MIP-9012549 and MIP-9203895, in part by the National Aeronautics and Space Administration (NASA) under Grant NAG-1-492, and in part by the Office of Naval Research under Grant N00014-91-J-1115. Recommended for acceptance by R. LeBlanc.

D. L. Kiskis was with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan 48109-2122. He is now with the Software Engineering Section, Center for Computer High-Assurance Systems, U.S. Naval Research Laboratory, Washington, DC, USA.

K. G. Shin is with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109-2122 USA; e-mail: kgshin@eecs.umich.edu.

IEEE Log Number 9405268.

we have developed an SWG to automatically generate all the programs for the distributed SW based on a compact high-level language specification. Such a specification can be created and debugged much more efficiently.

The SWG for HARTS has been completely implemented. It executes on a workstation that is separate from HARTS. The SWG produces C code for the SW and compiles it to produce an executable image for each processor. The executable image is then downloaded to HARTS and is executed there. While the SW is executing, performance and reliability measurements may be taken.

In this paper, we discuss the language that we have developed to specify SW's. This language is the Synthetic Workload Specification Language (SWSL). SWSL is based on a workload model that accurately describes the structure and behavior of real-time workloads executing on a distributed system. It specifies the timing characteristics of tasks, the interactions between tasks, and the internal structure and behavior of each task. Since SWSL is based on an abstract workload model, it is not specific to HARTS and can be used to specify SW's for other systems. SWSL provides a compact syntax to improve the ease with which experimenters may develop and alter workload specifications. This feature is particularly important if the target machine is composed of a large number of processors. A relatively small specification can be used to describe a workload consisting of many tasks. Finally, SWSL is designed to support the experimentation process. It contains features that allow the user to run a series of statistically independent experiments in an efficient manner.

This paper is organized as follows. In the next section, we describe the workload model upon which SWSL is based. Section III discusses important issues in specifying SW's that are addressed by SWSL. In Section IV, we define SWSL in the process of specifying an SW based on a submarine passive sonar system. Section V describes how SWSL is used by our SWG to produce SW's. We conclude with Section VI.

## II. Workload Model

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

In a real-time system, the value of a computation depends not only on the logical correctness of the results but also on the time when the results are produced. This definition describes a class of systems with characteristics that set them and their workloads apart from general purpose systems [5], [21], [31]. They are usually embedded in a larger system that performs a particular function. The real-time system serves as the controller computer for this larger system. The real-time system is designed to execute specific application software required to control the larger system. All tasks are predefined and their parameters are usually known *a priori*. The control activity consists of accepting frequent or continuously arriving inputs from sensors and, in response, producing output to actuators and/or display devices. These responses must occur soon enough after the input to meet the physical constraints of the system. The system must also accept inputs at random times due to operator commands and exceptional conditions. The hardware of the system may be distributed, consisting of a number of processors each connected to a variety of input-output (I-O) devices. Distributed systems exhibit great potential for high performance and high reliability, two properties that are essential for real-time systems.

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

Previous approaches to modeling workloads consisted of capturing the behavior of the workload using queueing networks or describing the workload in terms of a vector quantifying the workload characteristics [9]. However, the properties of a real-time workload are not accurately modeled by a queueing network or as a simple vector of workload parameters, because these techniques model average-case performance. Therefore, they cannot capture the details of the timing characteristics of the workload. To model a real-time workload, we must accurately describe the details of the workload that specifically influence the time-related aspects of the system. The model should express the tasks' timing, resource usage, and interaction characteristics. The timing characteristics include task execution times, deadlines, and scheduling parameters. The resource usage characteristics should include access priorities, preemption policies, and the quantity of the resources used along with the timing characteristics (e.g., pattern and duration) of that usage. Task interactions include both direct communication and resource sharing. Since standard queueing models and simple vectors of workload parameters are neither powerful nor expressive enough to model real-time workloads, a different, more expressive model is needed.

### A. Modeling Issues

We have constructed a model to accurately capture the structure and behavior of a real-time workload. The workload is described in terms of a data flow graph, a notation commonly used to specify software for distributed real-time systems. The model is a generalization of the rapid prototyping languages such as PSDL [19], and structured analysis (SA) notations such as ESML [3] and others [12], [33]. SA notations are commonly used in CASE tools to specify and analyze the requirements and structure of real-time software. The data flow model captures the basic aspects of the workload like parallelism and interactions between tasks, and it allows for modeling at multiple levels of abstraction. These features provide a generality that makes our model flexible and thus more widely applicable. So, it is capable of modeling the features of a number of SA, rapid prototyping, and other notations like those in [11], [20], [24], and can be used to describe a wide range of real-time workloads that have been specified using these notations. Our model extends these notations to specify the timing and resource usage properties of the workload.

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

- At the time a prototype system is ready for evaluation, it is likely that the system designers will have only a high-level specification of the proposed application software, e.g., the SA model. This model will generally be a good approximation of the structure of the workload [12]. Thus, by using a similar model for our SW, we can produce an SW that will closely approximate the structure and behavior of the proposed software. The experimental evaluations performed using this SW will then provide useful and meaningful results. Similarly, developers of experimental systems can make use of published workload specifications [16], [18], [22], [35] to produce representative SW's to be used to evaluate their systems.
- Since the workload is modeled at a high level of abstraction, the model is system-independent. The model does not contain any information that is particular to a given hardware architecture or operating system. Therefore, a workload specified using our model is portable and may be used to comparatively evaluate different systems.
- As real-time software becomes more complex, the use of structured methods of design the software will become widespread. the design process will be supported by computer-aided software engineering (CASE) tools [6], [25]. Our approach allows the SWG to become an integral part of a CASE tool. A number of CASE tools use SA and similar notations. Hence, high-level software designs created by CASE tools can be translated to our model and used by the SWG to create SW's. The SW's thus produced will be akin to a rapid prototype. The difference is that though the rapid prototype is aimed at demonstrating the functionality of the software from the user's viewpoint, the SW is aimed at demonstrating the resource use behavior of the software from the system's viewpoint.

## B. Formal Definition

A *real-time workload* is defined as a 5-tuple $(T, S, R, F, D)$, where $T$ is a set of *transformations*, $S$ is a set of *stores*, $R$ is a set of *terminators*, $F$ is a set of flows, and $D$ represents data. These workload objects will be described in detail in the following sections. The graphical representation of all components are shown in Fig. 1. These symbols are taken directly from the graphical representation for ESML [3].

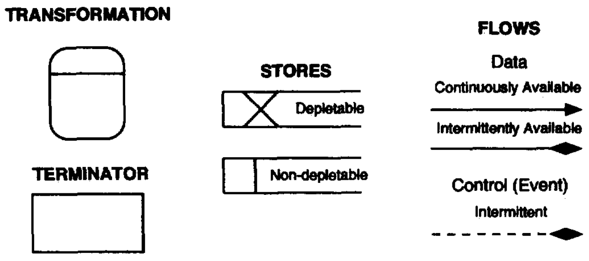


Fig. 1. Graphical representation of model components.

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

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

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

The timing and selection of inputs and outputs are determined by the internal structure and behavior of the transformation. Transformation behaviors are not restricted to the model of "trigger, compute, produce output" that is common to data flow specifications. Instead, transformations are free to perform inputs and outputs at any time during their execution. Based on their internal logic, they are also able to select whether to read a given input or produce a given output. This flexibility in defining task interactions is necessary when specifying SW's for real-time systems. If the SW is to be representative of real applications, the synthetic tasks must accurately reproduce the complex timing and resource-sharing dependencies between tasks. This accuracy cannot be obtained from a simple data flow model [34]. It requires the more detailed specifications allowed by this model.

The function specified by $\phi$ is defined based on the $D$-structures described by Ledgard and Marcotty [15]. The set of $D$-structures is a small functionally complete set of control constructs for programs. They consist of simple operations, composition of $D$-structures, a conditional control construct, and a loop construct. A simple operation is any computation, system call, or input or output statement, for example. These are the smallest units of execution in the model. Composition is the simple sequential execution of two $D$-structures. For two $D$-structures `s1` and `s2`, composition is represented as `s1; s2`. The conditional control construct is the "`if condition then s1 else s2`" construct. The looping construct is the "`while condition do s`" construct. With these constructs, all other control constructs may be realized [15].

The complete specification of a transformation depends on the system upon which it is to be implemented. Different operating systems require different information to create and schedule the implementation of the transformation. Therefore, the transformation specification includes a number of system-specific parameters, $\pi_i$. These parameters may include scheduling parameters, resource requirements functions for exception handling, and so forth. The number of $\pi_i$ parameters, and thus the value of $N$, depends on the target system. The model defines as many $\pi_i$ parameters as are needed to specify the implementation of transformations on a given target system.

*Stores:* *Stores* model all objects that can contain data. These objects include data structures, files, sockets, pipes, and so forth. A transformation passes data to another transformation by placing the data in a *store* from which the other transformation reads the data. Formally, we define $S = \{s \mid s = (I, O, p, \pi_1, \cdots, \pi_N)\}$, where $I$ is the set of inputs, $O$ and $p$ is a set of outputs, $p$ is a set of processors, and $\pi_1, \cdots, \pi_N$ is a set of $N$ system-specific parameters where $N$ depends on the target system. The $I$ and $O$ values have the same meaning as they have in the definition of $T$. The $\pi_i$ values define parameters required to specify the implementation of a data structure modeled by the store. They define its storage properties such as element size, storage policy, and access policy. $S$ may be divided into two disjoint subsets such that $S = S_d \cup S_n$, where $S_d$ is the set of *depletable* stores and $S_n$ is the set of *nondepletable* stores. Depletable stores represent objects such as stacks and queues where a data element is removed from the store when it is read. A nondepletable store represents an object like shared memory that retains the data value after a read. The reader receives a copy of the data without changing memory contents.

*Terminators:* Terminators serve as the interfaces between the workload and the environment. We define $R = R_i \cup R_o$, where $R_i$ is the set of input terminators and $R_o$ is the set of output terminators. $R_i = \{r \mid r = (O, p, \pi_1, \cdots, \pi_N)\}$ and $R_o = \{r \mid r = (I, p, \pi_1, \cdots, \pi_N)\}$, where $I$ is the set of inputs, $O$ is a set of outputs, $p$ is a set of processors, and $\pi_1, \cdots, \pi_N$ is a set of $N$ system-specific parameters. The $I, O$, and $p$ values have the same meaning as they have in the definition of $T$. The $\pi_i$ values define parameters required to specify the characteristics of the terminator. They specify the interface between the workload and the environment. They define the type of the interface, the size of the data elements that it handles, and the minimum sampling interval or minimum data acceptance interval.

The $R_i$ and $R_o$ terminators are referred to as *sources* and *sinks*, respectively. A source terminator represents a point where data are received by the workload from an external object. Typical examples of such objects are sensors and operator controls. Sink terminators represent locations where data or control signals are sent to an external object by the workload. Actuators and displays are examples of external objects that may be represented by sink terminators. Terminators may also be paired to represent resources such as external files or databases that have both inputs and outputs.

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

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

*Interconnection Rules:* Construction of a workload with our model is based on the construction of one using ESML [3]. The model construction rules are specified formally by the definitions of the flow types:

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

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

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

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

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

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

## III. Specification of Synthetic Workloads

Before discussing the details of SWSL, we first present the underlying concepts of its design.

### A. Abstraction

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

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

## B. Representativeness

By basing the SW on the workload model described in Section II, we improve the representativeness of the SW. To measure representativeness, we use a performance-based metric. By this metric, an SW is representative of a workload if the performance of the system (as measured by a set of performance indices) while executing the SW is the same as the performance while executing the workload [10]. However, "[e]xcept for ceratin cases···, this definition of [representativeness] does not directly suggest a method for designing an artificial workload" [10]. Given this observation, we use a structure-based method for constructing a representative SW. That is, the SW is specified and constructed such that its structure models that of the workload. Other researchers [2], [32] have successfully produced sufficiently accurate and flexible benchmark programs for uniprocessor systems by modeling the structure of the actual workload. We expect the technique to be successful for distributed systems as well. The structure-based representation is complemented by selecting the appropriate $\pi_i$ parameters for each object and assigning appropriate values to the parameters. These parameters determine the characteristics of the object as it is presented to the system. Parameters specify the resource requirements and the time-dependent behavior of the objects. By providing the SW with the same structure as the workload being modeled, and by tuning the parameters that determine the behavior, we are able to produce a representative SW. The level of representativeness may be measured by the performance-based metric. The ability of SWSL to produce representative SW's is demonstrated in [14].

## C. Flexibility

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

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

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

## D. Object Templates

The workload model defines each object uniquely. SWSL makes the specification of objects more compact by providing a simple mechanism whereby one can produce many instances of an object from a single *object template*. All instances of the object will have the same values of all the $\pi_i$ parameters. There are two uses for object templates. The first is to specify an object that represents a member of a class of objects with similar parameters. This technique is common in workload characterization [27] and has been used often to specify SW's, e.g., [1]. A set of $n$ parameters are selected to define the important characteristics of the workload tasks. For each task, these parameters are measured, and the task is plotted in the $n$-dimensional space defined by the parameter vector. A clustering analysis is performed to partition the tasks into groups with sufficiently similar characteristics. Then a small number of tasks from each group are selected to represent that group. These representative tasks are used as templates. The number of instances of the task that are produced is proportional to the size of the cluster being represented relative to the size of the entire workload. This technique reduces the number of task specifications that must be written, and thus makes the workload specification more compact. The second reason for using templates is to represent objects that are replicated for purposes of fault tolerance. In a fault-tolerant real-time system, multiple copies of an object will execute on separate processors. They perform the same calculations, and the results are combined via voting. Using this technique, the system can mask a given number of faulty processors.

### *E. Support for Experimentation*

Experimental design is supported through several SWSL features. First, we differentiate between the SW and the measurement mechanisms used to collect data in the evaluation. The only function of the SW is to serve as the workload for the target system; it does not provide any mechanisms for measuring performance. In this way, the SW is different from a benchmark program, which not only exercises the system but also measures the performance of the system while it is being exercised. The SW is designed to work harmoniously with performance evaluation mechanisms. Therefore, the user is free to choose any appropriate measurement mechanism. For example, if a software monitor that must be executed as a user task is being used, the monitor can be specified as the function for a transformation. The monitor will be compiled into the SW and function normally on the target system.

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

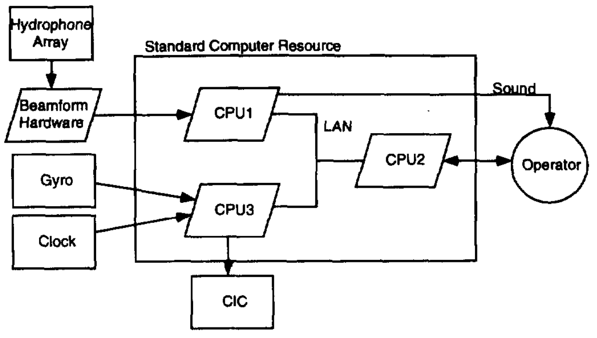


Fig. 2. Hardware architecture of passive sonar system.

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

SWSL has additional mechanisms to support experimentation that have been absent in previous SW's: reproducible experiments, independence of events within a given experiment, and statistically independent experiments. As described above, the SW reinitializes the experiment between runs. In addition, it simulates data-dependent activities stochastically. Each such activity makes use of a separate random number generator stream. This technique allows independent objects to exhibit reproducible, independent behavior. This feature is especially important when evaluating multiprocessor systems, where nondeterministic behavior is common. Sharing a random number stream would cause correlation between actions that would be irreproducible in a nondeterministic environment.

## IV. An Example SW Specification

We present the structure of SWSL in the context of an example. The example system is the submarine passive sonar system developed at IBM [23]. We have followed the specification in [23] as closely as possible. In places where the specification was vague or incomplete, we made assumptions and fabricated details based on the informal descriptions in the specification. We do not present the entire specification, but instead use components from the system to demonstrate the features of SWSL.

The passive sonar system assumes a hardware architecture as shown in Fig. 2. The hardware consists of three processors connected via a local area network. The peripheral hardware components (console, hydrophones, and so forth) are each connected to specific processors, and interaction with those components is through agents on those processors. The com-

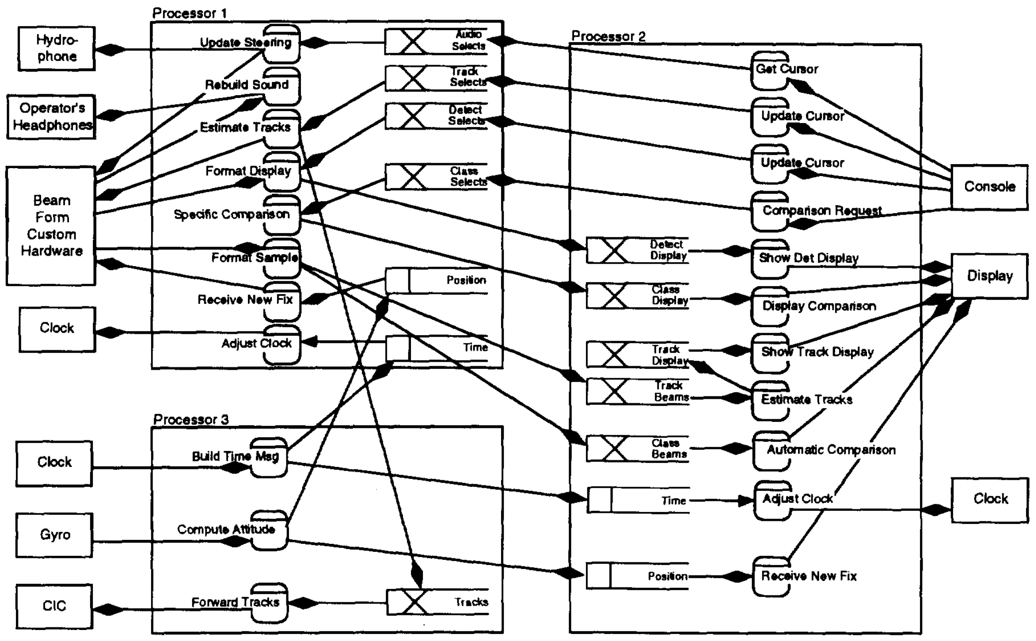


Fig. 3. Passive sonar specification.

munication and processing loads are distributed among the processors as shown in Fig. 3. The workload is divided into eight subsystems: Position Update, Timing Update, Signal Acquisition, Tracker Loop, Analysis Request, Detection Display Cursor, Track Display Cursor, and Audio Steering. Each subsystem consists of a number of communicating tasks.

The SW specification is apportioned among three different input files. The three files specify the task graph, the task functions, and the experimental parameters, respectively. Although each file has its own particular syntax, there are some constructs that are common to all the files. The SWSL files are divided into sections that contain common language features (declarations of locally and externally defined objects and constants) and sections that contain the specification of the SW. SWSL supports simple arithmetic expressions where operands are scalars, constants, or distributions. Distributions are used in expressions to indicate that the value is to be generated at run-time from a random number generator with the specified distribution and parameters. A library of random number generators with various distributions is provided with SWSL.

### *A. Graph File*

To demonstrate the details of the graph file, we specify only the Track Display Cursor subsystem, which is shown in Fig. 4. We concentrate on this single subsystem for the sake of brevity. The graph file defines the transformations, stores, and terminators, along with the $\phi$, $I$, $O$, $p$, and $\pi_i$ parameters for each objects. These parameters are discussed in the following sections.

The complete graph file for the Track Display Cursor subsystem is as follows.

```
GRAPH
EXTERNS
FUNC estimate_tracks_f;
FUNC update_cursor_tr_f;
FUNC forward_tracks_f;
PROC processor2;
PROC processor1;
PROC processor3;
CONSTANTS
/* Sonar parameters */
Nt = 5; / * Number of track beams * /
Na = 6; / * Number of audio beams * /
Nd = 50; / * Number of detection beams * /
Nc = 5; / * Number of analysis beams * /
BF = 4/ * Beamformer rate * /
track_selects capacity = 3;
track_size = 16 * Nt;
OBJECTS
TERM console1;
TRANS update_cursor_tr;
STORE track_selects;
TRANS estimate_tracks;
STORE tracks;
TERM beam_form_snk;
TRANS forward_tracks;
TERM cic;
DEFINITIONS
/ * TERM * / console1[
    OUTPUT = update_cursor_tr : discrete;
    PROCESSOR = processor2;
    TYPE = source;
    ELEMENT_SIZE = 48;
```

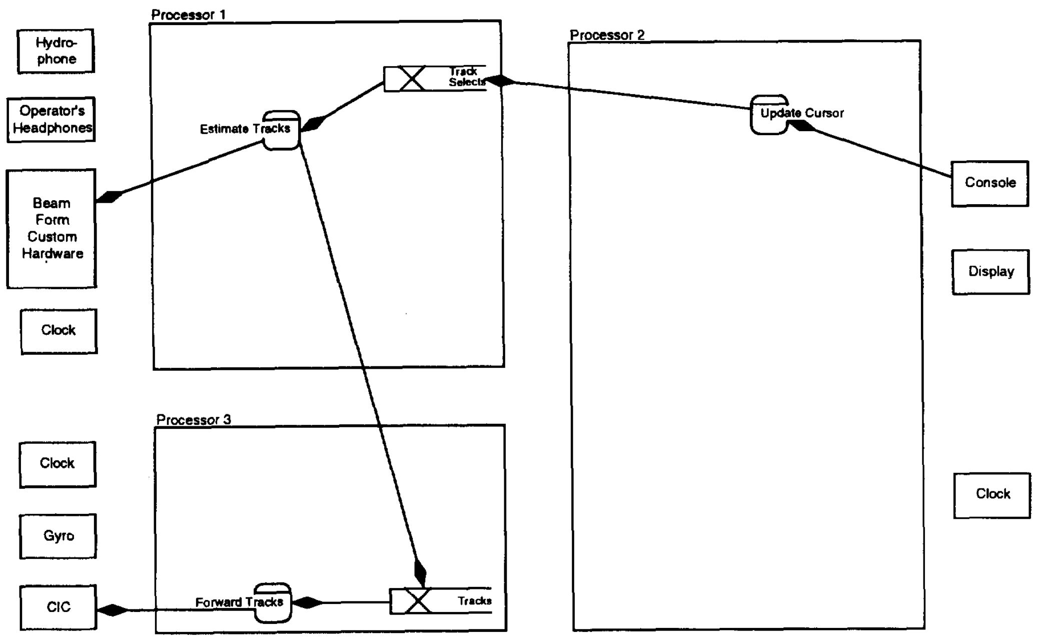


Fig. 4. Passive sonar specification: Track display cursor subsystem.

```
        RATE = 100;
        START_TIME = 1;
];
/ * TRANS * / update_cursor_tr[
        FUNCTION = update_cursor_tr_f();
        INPUT = consolel : discrete;
        OUTPUT = track_selects : discrete;
        PROCESSOR = processor2;
        PERIOD = 100, 75, 50;
        DEADLINE = 100;
        START_TIME = 1;
        ACTIVE = true;
        PRIORITY = 3;
];
/ * STORE * / track_selects[
        INPUT = update_cursor_tr : discrete;
        OUTPUT = estimate_tracks : discrete;
        PROCESSOR = processorl;
        TYPE = depletable;
        ELEMENT_SIZE = 24;
        CAPACITY = track_selects_capacity;
        POLICY = fifo;
];
/ * TRANS * / estimate_tracks[
        FUNCTION = estimate_tracks_f()
        INPUT = track_selects : discrete;
        OUTPUT = tracks : discrete;
        OUTPUT = beam_form_snk : discrete;
        PROCESSOR = processorl;
        PERIOD = 250;
        DEADLINE = 250;
        START_TIME = 1;
        ACTIVE = true;
        PRIORITY = 12;
];
/ * STORE * / tracks[
        INPUT = estimate_tracks : discrete;
        OUTPUT = forward_tracks : discrete;
        PROCESSOR = processor3;
        TYPE = depletable;
        ELEMENT_SIZE = track_size;
        CAPACITY = 1;
        POLICY = fifo;
];
/ * TERM * / beam_form_snk
        INPUT = estimate_tracks : discrete;
        PROCESSOR = processorl;
        TYPE = sink;
        ELEMENT_SIZE = track_size;
        RATE = 100;
        START_TIME = 1;
];
/ * TRANS * / forward_tracks[
        FUNCTIONS = forward_tracks_f();
        INPUT = tracks : discrete;
        OUTPUT = cic : discrete;
        PROCESSOR = processor3;
        PERIOD = 250;
        DEADLINE = 250;
        START_TIME = 1;
        ACTIVE = true;
        PRIORITY = 10;
];
/ * TERM * / cic[
    INPUT = forward_tracks : discrete;
    PROCESSOR = processor3;
    TYPE = sink;
    ELEMENT_SIZE = track_size;
    RATE = 250;
    START_TIME = 1;
];
```

*The $\phi$ Function:* The $\phi$ functions for transformations are declared using the `FUNCTION` parameter keyword followed by the list of names of the functions to be executed in each run. These functions are defined in the functions, file, which is discussed in Section IV-B.

*The $I, O$, and $p$ Parameters:* Specifications of $I, O$, and $p$ for each object use the parameters `INPUT`, `OUTPUT`, and `PROCESSOR`, respectively. To form a data flow graph, the objects in the workload must be connected by flows. A flow is specified implicitly using the `OUTPUT` parameter of the source object and the `INPUT` parameter of the destination object. Each `INPUT`, `OUTPUT` pair denotes a separate connection between the objects.

In software specification languages such as ESML [3], Ward and Mellor's transformation schema [33], and PSDL [19], all transformations must be connected by flows to other objects. A transformation that does not receive data from other objects is useless; it cannot do useful work. In contrast, in the SW, no transformation does useful work. All transformations only *behave* like they are doing something useful; they do not operate on real data. Therefore, we do not require that a transformation be connected to other objects. In some cases, the user may want to define a task that executes independently of other tasks. An example of this case is when specifying SW's to study scheduling algorithms without considering task interactions. The workload would consist of a number of independent tasks whose only workload characteristic was the amount of CPU time required. For this case, we do not require that the `INPUT` and `OUTPUT` parameters be defined. However, in most cases, the user will be interested in the effects of task interactions on system performance. Hence, most transformations will be connected to others and will have `INPUT` and `OUTPUT` parameters assigned to them.

The two parameters for flows are flow type and the size of the data elements that pass along the flow. Because of the model construction rules (see Section II-B), we can always determine the element size for a data flow from the components that it is connecting. A data flow must be attached at one end to either a store or a terminator, each of which has an `ELEMENT_SIZE` parameter. Hence, the flow can inherit this parameter from the object. Event flows carry no data, and thus require no size specification. Because the data element size can always be determined for a flow, we need to be able to specify only the type of the flow in the SWSL specification. Since flows have only a single parameter, we include the flow type in the specification of the `INPUT` and `OUTPUT` parameters for objects. The flow types are `DISCRETE`, `CONTINUOUS`, and `EVENT`, corresponding to intermittent data flows, continuous data flows, and event flows, respectively.

*The $\pi_i$ Parameters:* The $\pi_i$ parameters are specified in the body of each object definition. As shown in the specification of transformation `update_cursor_tr`, the `PERIOD` parameter has values of 100, 75, and 50. The $i$th value in the list is the value that the parameter is to take on for the $i$th run. If fewer values are listed than the number of runs, then the last value in the list will be used for its corresponding run and all subsequent runs. This feature is used to compactly specify parameter values that remain constant across runs. For example, the period of transformation `estimate_tracks` is 250 ms during each run. Therefore, the value needs to be specified only once, and that value will be used for all runs. Currently, all time values in SWSL are measured in milliseconds. Therefore, no indication of time unit is necessary in the specification.

The $\pi_i$ parameters for the transformations, stores, and terminators are shown in plain (nonbold) text in the SWSL listing. Transformation parameters indicate the transformation's scheduling requirements. The store parameters specify the type of data in the store and the access methods to be used. Stores represent all information repositories and data channels. Thus, the parameters have been selected such that they are orthogonal, and combinations of values may be used to represent different storage objects. Similarly, for terminators, we have chosen parameters to allow a range of terminator types.

The selection of $\pi_i$ parameters is not fixed. We selected these parameters to specify the system-dependent characteristics of real-time workloads on HARTS. If required, parameters may be added to the language by updating the list of recognized parameters in the SWG source code.

*Object Templates:* An object template is specified by using the `PROCESSOR` parameter in conjunction with the `INPUT` and `OUTPUT` parameters. By providing multiple values for the `PROCESSOR` parameter, the user specifies that an instance of an object is to be assigned to each of a number of processors. Each instance of the object will be assigned the same $\pi_i$ parameter values. The `INPUT` and `OUTPUT` parameters will differ, depending on the objects to which the copies of the object are connected. The connectivity of these objects is defined by using a special syntax for the `INPUT` and `OUTPUT` parameters. We present the syntax for object templates by considering the Timing Update subsystem. The Timing Update subsystem is shown in Fig. 5. In this subsystem, `build_time_msg` sends identical messages to the `time` stores on each of the other two processors. Identical `adjust_time` tasks read these messages and update their respective local clocks. A graph file containing only the specification of this subsystem is as follows.

```
GRAPH
EXTERNS
FUNC adjust_clock_f;
FUNC build_time_msg_f;
PROC processor2;
PROC processor1;
PROC processor3;
CONSTANTS
```

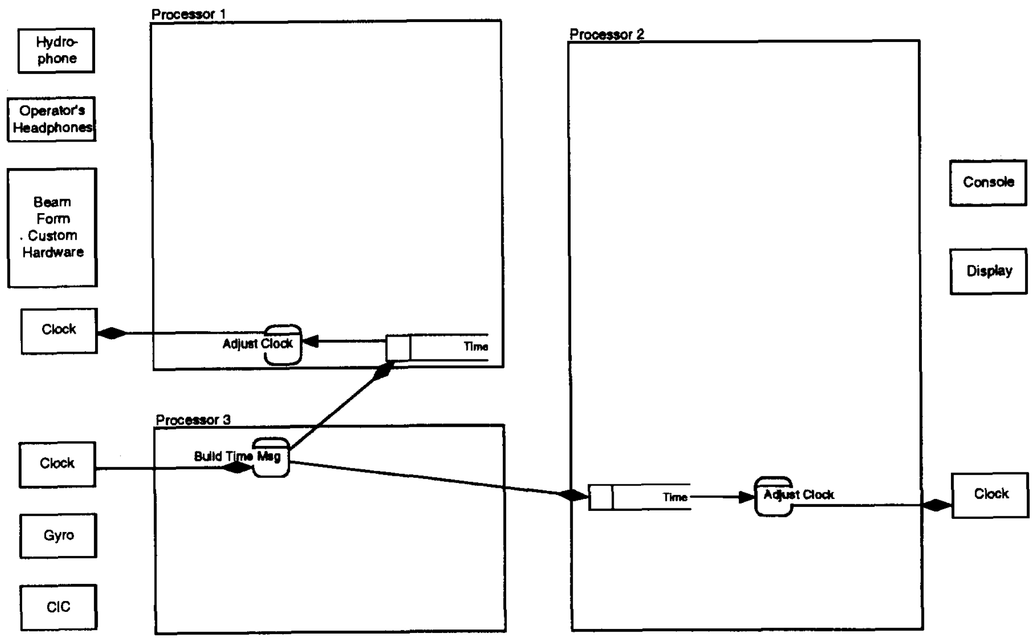


Fig. 5. Passive sonar specification: Timing Update subsystem.

```
clock_rate = 125;
OBJECTS
TERM master_clock;
TRANS build_time_msg;
STORE time;
TRANS adjust_clock;
TERM slave_clock;
DEFINITIONS
/* TERM */ master_clock[
    OUTPUT = build_time_msg: discrete;
    TYPE = source;
    ELEMENT_SIZE = 1;
    PROCESSOR = processor3;
    RATE = clock_rate;
    START_TIME = 1;
];
/ * TRANS * / build_time_msg[
    SPORADIC = 0;
    DEADLINE = clock_rate;
    FUNCTION = build_time_msg_f();
    START_TIME = 1;
    INPUT = master-clock : discrete;
    OUTPUT = time.processor1.0 : discrete;
    OUTPUT = time.processor2.0: discrete;
    PROCESSOR = processor3;
    ACTIVE = true;
    PRIORITY = 12;
];
/ * STORE * / time[
    TYPE = nondepletable;
    ELEMENT_SIZE = 8;
    INPUT = build_time_msg : discrete;
    OUTPUT = adjust_clock : continuous;
    PROCESSOR = processor1, processor2;
    CAPACITY = 1;
    POLICY = fifo;
];
/ * TRANS * / adjust_clock[
    PERIOD = clock_rate;
    DEADLINE = clock_rate;
    FUNCTION = adjust_clock_f();
    START_TIME = 1;
    INPUT = time : continuous;
    OUTPUT = slave_clock.processor1 : discrete |
      slave_clock.processor2 : discrete;
    PROCESOR = processor1, processor2;
    ACTIVE = true;
    PRIORITY = 9;
];
/ * TERM * / slave_clock[
    TYPE = sink;
    ELEMENT_SIZE = 1;
    INPUT = adjust_clock : discrete;
    PROCESSOR = processor1, processor2;
    RATE = clock_rate;
    START_TIME = 1;
];
```

In the following discussion, the object for which the inputs and outputs are being defined is referred to as the *current object*. The objects to which it is connected by flows from the inputs and outputs are referred to as the *connected objects*.

Transformation inputs and outputs are mapped one-to-one and in-order to function inputs and outputs. (See the discussion of the functions file below.) Therefore, the order in which inputs and outputs are specified is important. Furthermore, because SWSL supports object templates, the connected object may be one of many instances of an object. Hence, to accurately specify the connected object for an `OUTPUT`, for example, requires four pieces of information: the name of the connected object, the processor on which it is located, which `INPUT` of the connected object defines the other end of this flow, and the flow type.

Since the current object specification may also be a template, the specification of inputs and outputs is somewhat complex. However, simple, compact specifications are possible in most cases. The SWG makes some assumptions about how objects will be connected. It follows a set of rules based on these assumptions to fill in missing information in the simplified specification. At the minimum, the name of the connected object and the flow type are required. The other fields may be omitted if their values can be inferred by the SWG. The Timing Update subsystem example demonstrates both the full form of the syntax and the compact form. The specification of `build_time_msg` shows the complete form of the specification. It completely specifies that the flows connect to the copies of store `time` that are located on `processor1` and `processor2`, respectively, and that the flow is the first input (using zero-based counting) of `time`. `time` and `adjust_clock` use compact forms. The specification indicates that `time` is to be connected to the corresponding copy of `adjust_clock`. The objects are paired based on the order of the processors in the `PROCESSOR` parameter list.

In this case, objects that reside on the same processor are paired. The specification of the connection between `adjust_clock` and `slave_clock` demonstrates the full form of the syntax for connecting objects from templates. The vertical bars (|) separate specifications for the same `INPUT` or `OUTPUT` on different copies of the object. These specifications correspond positionwise to processor specifications in the `PROCESSOR` parameter list.

### *B. Functions File*

This file defines the $\phi$ functions for the transformations. An example of a function file follows.

```
FUNCTIONS
EXTERNS
OPER kinst;
OPER sread;
OPER swrite;
CONSTANTS
CODE
estimate_tracks_f{
    INPUT = selects: discrete;
    OUTPUT = tracks: discrete;
    OUTPUT = beam form: discrete;
    BEGIN
    LOOP track_selects_capacity
    {
        SWITCH
        {
            62 : {
                sread(selects, NOWAIT, 0);
                kinst(10 * Nt/BF);
                swrite(tracks, WAIT, 3, 10);
                swrite (beam_form, WAIT, 3, 10);
                };
                remaining: {
                        kinst(5 * Nt/BF);
                        swrite(tracks, WAIT, 3, 10);
                };
    };
    END;
};
```

This example shows the definition of the `estimate_tracks_f` function. The other functions would have to be provided in order for this to be a complete specification. The first part of the specification is a listing of the input and output flows of the function. These flows are given symbolic names that are used within the function. At compile time, the symbolic name is mapped to the corresponding `INPUT` or `OUTPUT` of the transformation that executes the function. In the example, `selects` will be mapped to the `track-selects` input of the `estimate_tracks` transformation. The code that is executed by the function is defined between the `BEGIN` and `END` keywords. During each periodic or sporadic invocation of the transformation, the code between the `BEGIN` and `END` keywords is executed exactly once.

The operations and control constructs from the workload model have been adapted for specifying SW's. Computation and communication are implemented with synthetic operations. Synthetic operations exercise specific resources in a predefined manner. The use of synthetic operations has been described in [2], [30], [32]. The synthetic operations are located in a library of operations. Synthetic operations are implemented as C functions. These functions are parameterized so the user can control their behavior. By defining them as functions, we hide the implementation details. Hence, the SW function specifications are made target system–independent. All system dependencies are contained in the implementation of the operations.

We have collected a number of synthetic operations for the library. Some of these were taken from the publicly available Bell Labs Benchmark suite and the dhrystone benchmark. They perform functions such as Ackerman's function, floating-point arithmetic, and word counts that exercise specific system functionalities. The operations are parameterized to control the number of iterations of each function. Additional operations may easily be added to the library.

In this example, `kinst()`, `sread()`, and `swrite()` are synthetic operations. A call to `kinst(n)` executes $n \times 1000$ integer operations. This synthetic operation is used to produce the desired computation load, which is specified in Kiloinstructions in [23]. The `sread` and `swrite` operations are particularly important. They are defined to be generic input and output operations. The primary parameter for these functions is the symbolic name of the input or output. The user need not specify any information about the object with which the function is communicating via the operation. The operations take information generated by the SWG for each input and output of the function and determine the appropriate system call(s) on the target system to use to perform the appropriate reading or writing operation. Therefore, they may be used in any function, regardless of the transformation that executes it and regardless of the objects to which the transformation is connected. These operations increase flexibility by introducing the capability for *plug-in* functions. During normal use of SWSL and the SWG, it is anticipated that the user will code a number of functions with different behaviors representing different types of tasks. These functions will be "plugged in" to transformations as needed for the particular application. Thus, SW's can be quickly constructed from components with known characteristics.

Control flow within a function is achieved by using sequential execution and the `LOOP` and `SWITCH` constructs. Both of these constructs are demonstated in the example Functions file. LOOP is an adaptation of the `while-do` loop in the workload model. It is a single-entry, single-exit looping construct. The parameter after the LOOP keyword specifies the loop count. The LOOP may be made to execute a constant number of times for each run, or it may execute a random number of times by specifying a distribution function as the loop count. By looping a random number of times, the loop simulates the behavior of data-dependent loops in the workload being modeled.

Branching is accomplished using the `SWITCH` construct. It is a generalization of the `if-then-else` construct in the workload model. It is derived from the `SWITCH` operation in [30]. In the `SWITCH` statement, the user specifies alternate blocks of code to be executed. Probability values are assigned to each block. Each time the `SWITCH` is executed, one block is selected at random based on the probability value assigned to the block. By branching probabilistically, it simulates the behavior of real applications that branch based on data values. The example specifies that 62% of the time the first block is to be executed, and that for the remaining percentage of the time, the second block is to be executed. If the percentage do not add to 100% and there is no `remaining` case, then the remaining percentage of the time, no operations are performed.

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

### C. Experiment File

The graph and functions files define the structure of the workload. The experiment file defines the behavior of the SW in the context of an evaluation experiment. Suppose that we are conducting an experiment to measure the effect on system performance causes by the different `PERIOD` values defined for transformation `update_cursor_tr`. We would use the following experiment file.

```
EXPERIMENT
CONSTANTS
Runs = 3;
PARAMETERS
processor1[
    TIMING = true;
    TIMELIMIT = 2000;
    SEED = 98752342;
    SEED_RESET = TRUE;
];
DEFAULT[
    TIMING = true;
    TIMELIMIT = 2000;
    SEED = 12348712;
    SEED_RESET = TRUE;
];
```

We want to execute three runs, so we set the value of the `Runs` constant to 3. This parameter applies to all processors in the experiment. The other experiment parameters are defined on a perprocessor basis, to account for differences between processors. The first set of parameters apply to `processor1` only. The `DEFAULT` entry defines the parameters for all processors for which there is no explicit entry. In this example, we are specifying that `processor1` is to use a different seed for its random number generator than is used by the other processors.

Two of the experiment parameters indicate whether the experiment is to be timed and define the duration of the execution interval. The other parameters provide control of the statistical properties of the stochastic behavior of the SW. They define the seeds of the random number generators. Each value of the `SEED` parameter defines a separate random number generator with the specified initial seed value. Each distribution function in the graph and function files can calculate its values from a separate stream. In this way, consecutive values generated by the distribution will be independent. The result is statistically independent events in the SW's execution. The `SEED_RESET` parameter may be used to reset the seed values at the beginning of each run. In this way, the behavior of individual streams may be reproduced. As with the $\pi_i$ parameters in the graph file, a list of values for the experiment parameters indicates the values to be used for each run.

## V. Synthetic Workload Generation

We have designed and implemented the SWG that compiles SWSL. The SWG completely automates the generation of SW's. The synthetic workload generation process is shown in Fig. 6. The SWG compiles the SWSL graph file to produce an internal representation of the task graph. It checks the graph for compliance to the connection rules. It then processes the inputs and outputs of the components to expand any specifications that use the simplified specification notation. Next it compiles the experiment file. Then it compiles the functions file and produces C language code for each function. While producing these files, it uses information from the task graph to expand the input and output labels in the functions. Then it generates files containing tables of the parameter values for the objects on each processor. The files for the SW on each processor are then compiled and linked to create an executable image. Compilation of the SW files is controlled by the SWG, which

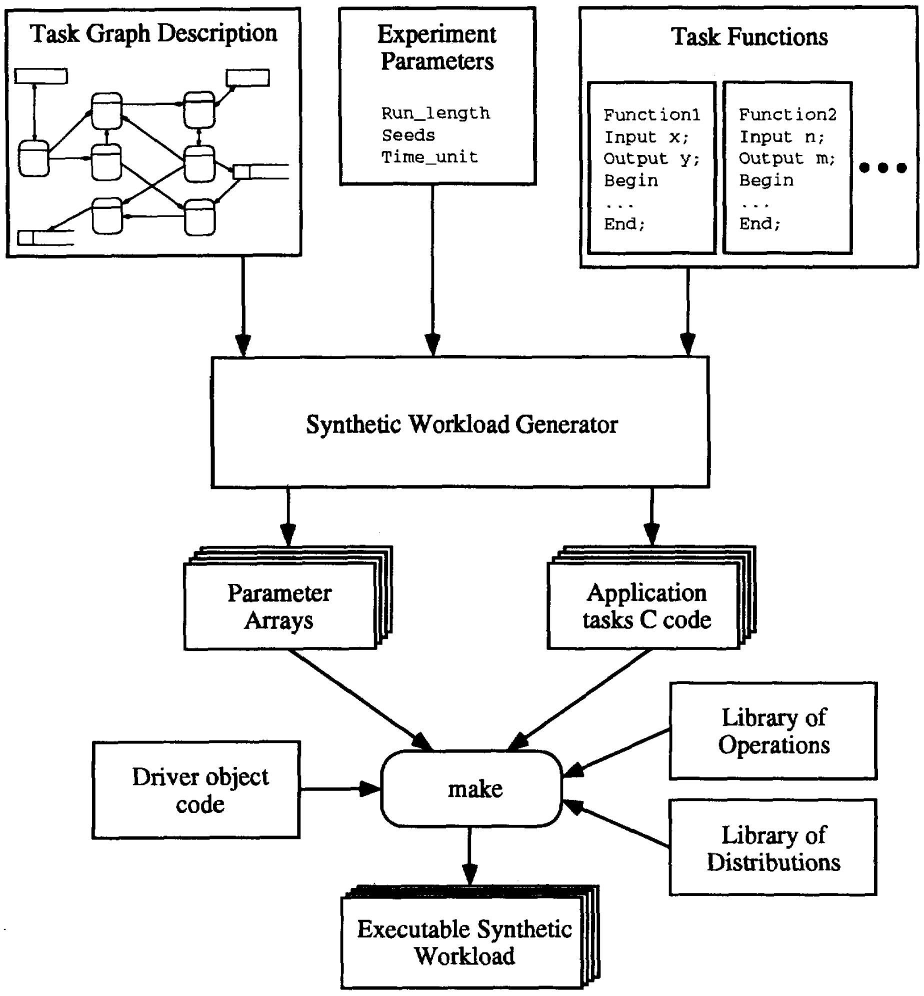


Fig. 6. Synthetic workload generation.

uses the processor assignment information from the graph file to direct the *make* utility.

## VI. Summary and Prospects for Future Work

SWSL is a language for specifying synthetic workloads for distributed real-time systems. It is designed to be easy to use while providing compact and flexible specifications. It is based on a workload model that makes it compatible with commonly used software specification notations. Hence, it is capable of accurately modeling real workloads. The language includes facilities for experiment support such as support for multiple runs, replication of objects, multiple random number generator streams, and modeling of stochastic behavior.

The SWG has been developed and implemented. It supports all features described in this paper. It has been used to produce SW's for the initial testing and evaluation of HARTS. In another study, a robot control system was modeled. An SWSL specification was created based on this model, and an SW was generated for the robot control computer. Based on statistical comparison, the SW was shown to be representative of the actual control software. The SWG was also used to evaluate the performance of the communication subsystem of HARTS under different workloads [14].

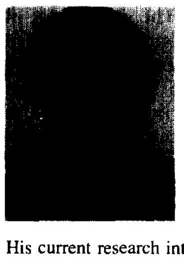
Enhancements to the language are being considered. One is to simplify the naming conventions for objects. In particular, we would devise a naming scheme that would allow multiple instantiations of an object from a template to reside on the same processor. This feature would reduce the need for redundant specifications in some instances. We also need to expand the number of synthetic operations that are available to the user.

## References

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

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

[3] W. Bruyn, R. Jensen, D. Deskar, and P. Ward, "ESML: An extended systems modeling language based on the data flow diagram," *ACM Software Eng. Notes*, vol. 13, pp. 58–67, 1988.
[4] W. Buchholz, "A synthetic job for measuring system performance," *IBM Syst. J.*, vol. 8, no. 4, pp. 309–318, 1969.
[5] R.M. Cohen, "Formal specifications for real-time systems," in *Proc. Texas Conf. Computing Syst.* , 1978, pp. 1.1–1.8.
[6] H. Falk, "CASE tools emerge to handle real-time systems," *Comput. Design*, vol. 27, pp. 53–74, Jan. 1988.
[7] F. Feather, D. Siewiorek, and Z. Segall, "Validation of a fault-tolerant multiprocessor: Synthetic workload implementation," in *Proc. Int. Conf. on Distrib. Computing Syst.*, 1986, pp. 303–312.
[8] D. Ferrari, "Workload characterization and selection in computer performance measurement," *IEEE Comput.*, vol. 5, no. 4, pp. 18–24, July 1972.
[9] ——, *Computer Systems Performance Evaluation* . Englewood Cliffs, NJ: Prentice-Hall, 1978.
[10] ——, "On the foundations of artificial workload design," in *Proc. 1984 ACM SIG-METRICS Conf. Measurement and Modeling Comput. Sys.*, 1984, pp. 8–14.
[11] H. Gomaa, "A software design method for real-time systems," *Commun. ACM*, vol. 27, no. 9, pp. 938–949, Sept. 1984.
[12] D.J. Hatley and I.A. Pribhai, *Strategies for Real-Time System Specification.* New York: Dorset House, 1987.
[13] D.L. Kiskis and K.G. Shin, "A synthetic workload for real-time systems," in *Proc. 7th IEEE Workshop on Real-Time Operating Syst. and Software*, 1990, pp. 77–81.
[14] D.L. Kiskis, "Generation of synthetic workloads for distributed real-time computing systems," Ph.D. dissertation, Univ. of Michigan, Ann Arbor, USA, Aug. 1992.
[15] H.F. Ledgard and M. Marcotty, "A genealogy of control structures," *Commun. ACM*, vol. 18, no. 11, pp. 629–639, Nov. 1975.
[16] C.D. Locke, "Generic avionic software," IBM Syst. Integration Div., DRAFT, Oct. 1988.
[17] H.C. Lucas, "Synthetic program specifications for performance evaluation," in *Proc. ACM Ann. Conf.*, 1972, pp. 1041–1058.
[18] G.A. Ludgate, B. Haley, L. Lee, and Y.N. Miles, "The use of structured analysis and design in the engineering of the TRIUMF data acquisition and analysis system," *IEEE Trans. Nucl. Sci.*, vol. NS-34, no. 1, pp. 157–161, Feb. 1987.
[19] Luqi, V. Berzins, and R.T. Yeh, "A prototyping language for real-time software," *IEEE Trans. Software Eng.*, vol. 14, pp. 1409–1423, Oct. 1988.
[20] H.G. Mendelbaum and D. Finkelman, "CASDA: Synthesized graphic design of real-time system," *IEEE Comput. Graphics and Applic.* , vol. 9, pp. 40–46, Jan. 1989.
[21] A.K. Mok, "The design of real-time programming systems based on process models," in *Proc. Real- Time Syst. Symp.*, 1984, pp. 5–17.
[22] J. Molini, S. Maimon, and P. Watson, "Real time distributed system studies/scenarios," in *ONR 3rd Ann. Workshop: Foundations of Real-Time Computing*, 1990, pp. 187–209.
[23] J.J. Molini, S.K. Maimon, and P.H. Watson, "Real-time system scenarios," in *Proc. Real-Time Syst. Symp.*, 1990, pp. 214–225.
[24] A.H. Muntz and R.W. Lichota, "A requirements specification method for adaptive real-time systems," in *Proc. Real-Time Syst. Symp.*, 1991, pp. 264–273.
[25] P.W. Oman, "CASE analysis and design tools," *IEEE Software*, vol. 7, pp. 37–43, May 1990.
[26] H.D. Schwetman and J.C. Brown, "An experimental study of computer system performance," in *Proc. ACM Ann. Conf.*, 1972, pp. 693–703.
[27] G. Serazzi, Ed., *Workload Characterization of Computer Systems and Computer Networks.* Amsterdam, Netherlands: North-Holland, 1985.
[28] K.G. Shin, "HARTS: A distributed real-time architecture," *IEEE Comput.*, vol. 24, pp. 25–35, May 1991.
[29] A. Singh, "Pegasus: A controllable, interactive, workload generator for multiprocessors," M.S. thesis, Carnegie Mellon Univ., Pittsburgh, PA, USA, Dec. 1981.
[30] A. Singh and Z. Segall, "Synthetic workload generation for experimentation with multiprocessors," in *Proc. Int. Conf. Distrib. Computing Syst.*, 1982, pp. 778–785.
[31] J.A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *IEEE Comput.*, vol. 21, pp. 10–19, Oct. 1988.
[32] R.E. Walters, "Benchmark techniques: A constructive approach," *Comput. J.*, vol. 19, no. 1, pp. 50–55, Feb. 1976.
[33] P.T. Ward and S.J. Mellor, *Structured Development for Real-Time Systems*, vols. 1–3. Englewood Cliffs, NJ: Yourdon, 1986.
[34] S.M. White and J.Z. Lavi, "Embedded computer system requirements workshop," *IEEE Comput.*, vol. 18, no. 4, pp. 67–70, Apr. 1985.
[35] B.E. Withers, D.C. Rich, D.S. Lowman, and R.C. Buckland, "Software requirements: Guidance and control software development specification," NASA Contractor Rep. 182058, Research Triangle Inst., Durham, NC, USA, June 1990.
[36] D.C. Wood and E.H. Forman, "Throughput measurement using a synthetic job stream," in *AFIPS Fall Joint Comput. Conf.* , vol. 39, pp. 51–55, Nov. 1971.
[37] M.H. Woodbury, "Workload characterization of real-time computing systems," Ph.D. dissertation, Univ. of Michigan, Ann Arbor, USA, Aug. 1988.



**D.L. Kiskis** received the B.S. degree (with highest honors) in mathematics and computer science from Denison University, Granville, OH, USA, in 1986, and the M.S. and Ph.D. degrees in computer science and engineering from the University of Michigan, Ann Arbor, MI, USA, in 1989 and 1992, respectively.

Since September 1992, he has been working in the Software Engineering Section of the Center for Computer High-Assurance Systems at the U.S. Naval Research Laboratory, Washington, DC, USA. His current research interests include distributed real-time systems, workload characterization, requirements specification, and software engineering processes.

Dr. Kiskis is a member of the IEEE Computer Society, Sigma Xi, and Phi Beta Kappa.

**K.G. Shin** (S'75–M'78–SM'83–F'92) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Republic of Korea, in 1970, and the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, USA, in 1976 and 1978, respectively.

He is a Professor of Electrical Engineering and Computer Science for the Computer Science and Engineering Division, The University of Michigan, Ann Arbor, MI, USA. He also chaired the CSE Division for three years beginning 1991. From 1978 to 1982, he was on the faculty of Rensselaer Polytechnic Institute, Troy, NY, USA. He has held visiting positions at the U.S. Airforce Flight Dynamics Laboratory, AT&T Bell Laboratories, Computer Science Division, within the Department of Electrical Engineering and Computer Science, University of California at Berkeley, USA, and International Computer Science Institute, Berkeley, CA, USA. He has also been applying the basic research results of real-time computing to manufacturing-related applications ranging from the control of robots and machine tools to the development of open architectures for manufacturing equipment and processes. Recently, he has initiated research on the open-architecture Information Base for machine tool controllers.

Dr. Shin has authored or coauthored over 270 technical papers (about 130 of these in archival journals) and several book chapters in the area of distributed real-time computing and control, fault-tolerant computing, computer architecture, robotics and automation, and intelligent manufacturing. In 1987, he received the Outstanding IEEE Transactions on Automatic Control Paper Award for a paper on robot trajectory planning. In 1989, he also received the Research Excellence Award from The University of Michigan. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are currently building a 19-node hexagonal mesh multicomputer, called HARTS, to validate various architectures and analytic results in the area of distributed real-time computing. He was the Program Chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the General Chairman of the 1987 RTSS, the Guest Editor of the 1987 August special issue of IEEE TRANSACTIONSON COMPUTERS on Real-Time Systems, a Program Co-Chair for the 1992 *International Conference on Parallel Processing*, and served numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems during 1991–93, is a Distinguished Visitor of the Computer Society of the IEEE, an Editor of IEEE TRANSACTIONSON PARALLELAND DISTRIBUTED COMPUTING, and an Area Editor on*International Journal of Time-Critical Computing Systems.*