

# A Synthetic Workload for a Distributed Real-time System \*

DANIEL L. KISKIS

dlk@umich.edu

KANG G. SHIN

kgshin@umich.edu

*Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science,  
The University of Michigan, Ann Arbor, Michigan 48109–2122*

**Abstract.** In this paper, we describe the design and implementation of a synthetic workload (SW) for a distributed real-time system. A SW is a set of parameterized synthetic or artificial programs which serve as the workload for a system under study. The parameterized nature of the programs allows the user to change their behavior to create different resource demands on the system. The SW is easy to use, flexible, and can be representative of a real-time workload. The SW consists of a driver and a set of synthetic tasks. The synthetic tasks are generated by a synthetic workload generator (SWG) from the user's specification written in SWSL, a synthetic workload specification language. We describe the design goals of our SW and discuss its software structure and how it meets these goals.

## 1. Introduction

The field of real-time systems is aimed at providing system support for real-time applications. The hardware, operating system, and network must be able to support the resource demands of the application. As systems are developed, the developers must experimentally evaluate the system's ability to meet its performance requirements. During evaluation, the system's performance is characterized using an appropriate set of performance indices, such as CPU utilization, task scheduling delay, message latency, or number of deadlines missed. The values of these indices depend on both the system and its workload.

The *workload* is the set of inputs to a computer system. It includes the application tasks, their input data, and the user commands. The resource demands produced by the workload are the *workload characteristics*. Many performance indices vary as a function of one or more workload characteristics. Hence, to fully characterize a performance index, one would like to be able to selectively alter the value of the workload characteristics. One tool which provides this ability is a *synthetic workload* (SW), an executable workload model (Ferrari, 1978) which consists of a set of artificial programs which produce controllable demands on the resources of the system. The programs are parameterized such that the user may control the workload characteristics. A SW differs from a traditional benchmark program in that it is tunable and thus can be used to pro-

---

\* The work reported in this report was supported in part by the Office of Naval Research under Contract No. N00014–91–J–1115, and the National Science Foundation under Grant No. MIP–9203895. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the view of the ONR or the NSF.

duce a wide range of workload conditions. By contrast, a benchmark program exercises the system in a very limited manner.

A SW should be representative, flexible, and simple to construct (Ferrari, 1978). The desire to attain these properties has guided SW design since the first SW was developed by Buchholz (Buchholz, 1969) for data processing systems. Buchholz's SW consisted of a single synthetic job. It was designed to model a commercial file update system common to business applications. The parameters specified the amount of data to be processed and the amount of processing to be performed on each data element. Various improvements were made to this workload by others (Schwetman and Brown, 1972, Sreenivasan and Kleinman, 1974, Wood and Forman, 1971). All attempted to increase the degree which the SW represented a real workload by adding parameters to the workload and/or by using multiple copies of the synthetic job, each with different parameter values. A variation of this latter concept was proposed by Lucas (Lucas, 1972). Lucas' system used a job mix which varied both the parameters of the synthetic jobs and their structure. Different job structures would be chosen to represent different types of applications. Since then, the design of SW's has been studied extensively by Ferrari (Ferrari, 1978, Ferrari, 1981, Ferrari, 1984).

To date, most SW's have been developed for general-purpose computing systems. They do not attempt to be representative of real-time workloads. Real-time workloads' characteristics are different from other workloads because real-time computers are often part of an embedded control system. Hence, real-time workloads typically contain a large proportion of periodic tasks whose periods are dependent on the required sampling interval or output rate of the system being controlled. They also contain a number of asynchronous tasks which execute in response to random events. Both periodic and asynchronous tasks may have hard deadlines to meet.

SW's for real-time systems are scarce. An early example was the SW for NASA's Fault-Tolerant Multiprocessor (FTMP), which is discussed in (Feather, 1984, Feather et al., 1986).. FTMP's SW was designed to exercise the system and perform a limited number of timing measurements. It defined the workload as a number of periodic tasks divided into three *rate groups*. A rate group was a collection of periodic tasks with the same period which were invoked at the same time. The periods of the rate groups were aligned at *major cycle* boundaries. A major cycle was the least common multiple of the lengths of the periods. Thus, at the beginning of each major cycle, all tasks were invoked simultaneously. The deadline for each task was equal to the length of its period. This SW was designed specifically for FTMP. Its applicability for use on other systems was restricted by its inflexibility. It had a fixed synthetic program structure, fixed deadline policy, and lack of aperiodic tasks.

Support for real-time SW's has also been built into Scheduler 1-2-3, a schedulability analyzer developed at Carnegie-Mellon University (Tokuda and Kotera, 1988). Scheduler 1-2-3 is capable of producing workload tables as a part of its schedulability analysis. The main workload parameters in the table are the period, priority, and phase (alignment of the periods) of the tasks. These tables can then be included into the SW which is used to test the ART Real-Time Testbed.

We describe in this paper our efforts to remedy the lack of SW's for real-time systems. We have designed and implemented a SW for a distributed real-time system. It has been developed to execute on the Hexagonal Architecture for Real-Time Systems (HARTS) being built at the Real-Time Computing Laboratory (RTCL) (Shin, 1991). While it has been developed for this target system, the design is general enough that it should be portable to other distributed multiprocessor systems.

We have developed a model of real-time workloads based on a common software specification notation. We then created SWSL (Kiskis, 1994), a synthetic workload specification language, to allow users to specify SWs using the constructs in the model. A SWSL specification is compiled by the synthetic workload generator (SWG) to produce an executable SW. The workload model and SWSL are defined in (Kiskis, 1994); only a brief outline of them is presented here for completeness. In this paper, we describe how the SW is implemented and how it supports the constructs in the abstract SWSL specification.

The paper is organized as follows. Section 2 describes the HARTS architecture for which the SW has been developed. We discuss the process of specifying and generating SWs in Section 3. In Section 4, we briefly describe how SWs are used in experimentation. In Section 5 the structure of the SW is outlined, and its implementation and use are described. In Section 6, we present empirical measurements of the overhead incurred by the control portions of the SW. The paper concludes with Section 7.

## 2. Target System

The facilities in the RTCL provide the means to demonstrate and verify basic research results in a carefully controlled realistic environment. The SW was initially constructed for a 19-node version of HARTS (Shin, 1991). Each HARTS node is a shared memory multiprocessor formed by up to three Motorola 680x0 microprocessors which serve as the *application processors* (AP's) for the node. The architecture of a node and its relationship to the development environment are shown in Fig. 1. The multiprocessor nodes are to be connected via a wrapped hexagonal mesh network. A hexagonal mesh is a 6-regular homogeneous graph. The AP's are connected to the network by custom-designed communication hardware, called the *network processor* (NP). The nodes are connected by a dedicated Ethernet to each other and to a workstation. The workstation serves as the console for the HARTS nodes.

A first version of the operating system for HARTS, called HARTOS, has already been completed and is operational (Shin et al., 1992). It is built upon the real-time kernel pSOS (Software Components, 1986) which provides a number of basic real-time kernel functions. The computation model for pSOS is process-based and supports preemptive, priority-based scheduling of processes. Within a priority class, execution may be FCFS, round-robin by timeslice, or manual round-robin. pSOS does not support periodic scheduling of processes nor does it enforce deadlines. Communication between processes is via *event* signalling and message passing through mailbox structures called *message exchanges*.

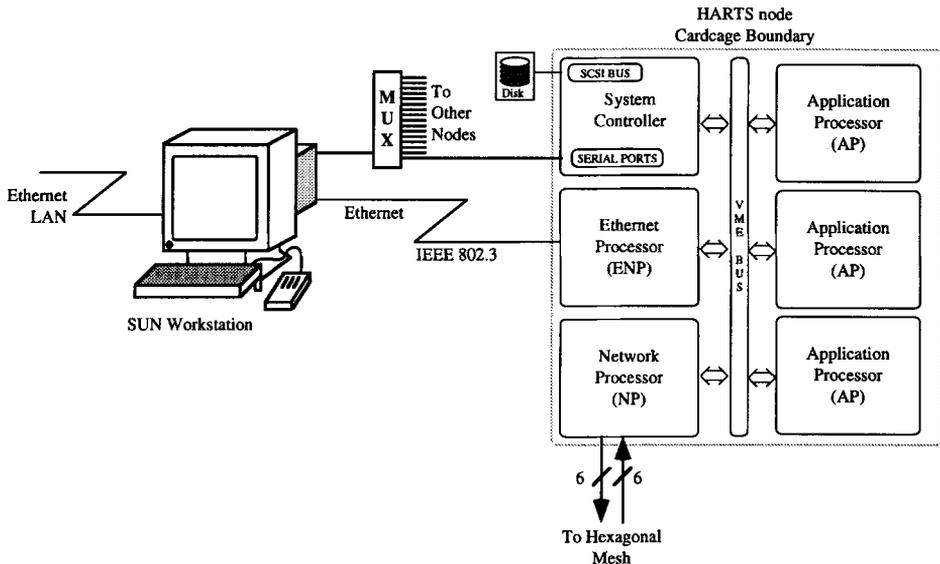


Figure 1. HARTS node architecture and operating environment.

A major objective of HARTOS is to provide internode and intranode communications by exploiting the NP in each HARTS node. HARTOS extends the pSOS message passing primitives for use between processes on different processors which are located either in the same node or different nodes. While the hexagonal mesh network is being designed and built, the HARTOS communication software executes on the Ethernet processor (ENP). Most of the operating system and software developed with the Ethernet is expected to be portable to the hexagonal interconnection network.

### 3. Specifying and Generating a Synthetic Workload

#### 3.1. The Workload Model

In this section, we give an overview of our model of real-time workloads. A detailed description is provided in (Kiskis, 1994). We model the workload of a real-time system as a parameterized dataflow graph. The notation for the dataflow graph is borrowed from ESML (Bruyn et al., 1988). ESML is a widely accepted notation for the specification of real-time software systems. It is a high-level notation which is independent of the target architecture. By basing our workload model on this notation, we aim to maintain the same generality and platform independence. We should be able to model a wide range of disparate workloads, and the SW which implements the model should be portable to a wide range of platforms.

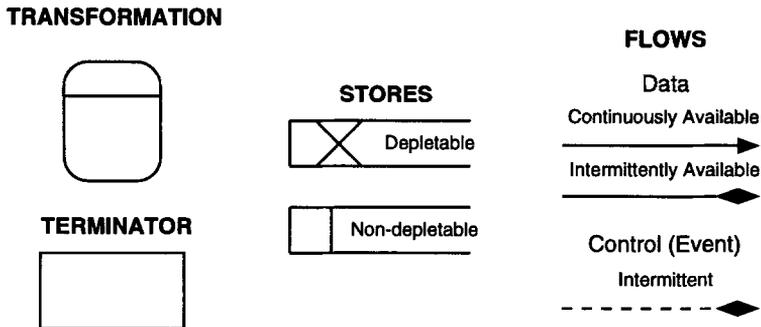


Figure 2. Data flow symbols.

The model represents the workload as consisting of transformations, stores, terminators, and flows. The symbols used for depicting these objects are shown in Figure 2. We have extended the ESMML notation by defining parameters to specify the objects' characteristics. Parameters may be added or deleted for different target systems with, e.g., different scheduling policies or interprocess communication mechanisms. The parameters described here are those which are appropriate for HARTS.

Tasks are modeled as *transformations*. Transformations operate on data from their inputs and produce data on their outputs. The parameters for transformations are shown in Table 1. They define the scheduling and resource usage behavior of the task. Of particular importance is the *FUNCTION* parameter. It defines the function which is executed by the transformation. This function determines the internal behavior of the transformation. The control flow of the function is modeled using the D-structures described by Ledgard and Marcotty (Ledgard and Marcotty, 1975). The D-structures are a small functionally-complete set of control constructs for programs. They consist of simple operations (assignments, computations, system calls, or input or output statements), composition of D-structures, a conditional branching construct, and a loop construct.

Data structures are modeled as *stores*. Stores may be *depletable* or *nondepletable*. Data placed in depletable stores is removed when it is read. Data in nondepletable stores remains in the store until it is overwritten. The parameters for stores are listed in Table 2.

Interfaces between the workload and external devices are model using *terminators*. *Source terminators* generate data or signals to the workload from objects outside the workload. *Sink terminators* receive data from the workload and pass it to objects outside the workload. The parameters for terminators are listed in Table 3.

All the workload components listed above are connected by flows. Flows are specified using the *INPUT* and *OUTPUT* parameters of the transformations, stores, and terminators that they connect. There are two types of flows, *data flows* and *control flows*. Data flows carry data between components. Data is available on *discrete* flows only at specific instances of time. Data is always available on *continuous* flows. Control (or *event*) flows carry control signals. Data is differentiated from control signals in that for data,

Table 1. Transformation parameters.

Parameter Name	Description
PERIOD	The period of a periodic transformation.
START_TIME	The time of the first invocation of the transformation.
SPORADIC	The function which returns the time between invocations for a sporadic transformation. The default of 0 may be used for transformations which are triggered by the availability of data or control signals.
DEADLINE	The deadline of the transformation, either a scalar value or distribution name or 0 to indicate no deadline.
FUNCTION	The name of the function which is executed by the transformation.
PRIORITY	The priority of the transformation.
INPUT	The name of the store, terminator, or transformation providing input.
OUTPUT	The name of the store, terminator, or transformation accepting output.

Table 2. Store parameters.

Parameter Name	Description
TYPE	The type of the store: DEPLETABLE or NONDEPLETABLE.
ELEMENT	The size (in bytes) of each element in the store.
CAPACITY	The storage capacity of the store measured in number of elements.
ACCESS	The access policy for the store: EXCLUSIVE or ALL.
POLICY	The storage policy for a depletable store, either FIFO, LIFO, or PRIORITY.
INPUT	The name of the transformation providing input.
OUTPUT	The name of the transformation accepting output.

Table 3. Terminator parameters.

Parameter Name	Description
TYPE	The type of the terminator: SOURCE or SINK.
ELEMENT	The size (in bytes) of each element generated or accepted by the terminator. May be a constant or variable value.
RATE	The time between data arrivals at a source terminator or the minimum time between data acceptances for a sink terminator. Either a constant value or a value taken from a specified distribution.
START_TIME	The time that the terminator is to begin producing or receiving data.
ACCESS	The access policy for the terminator: EXCLUSIVE or ALL.
INPUT	The name of the transformation providing input.
OUTPUT	The name of the transformation accepting output.

the value is important. For control signals, only the presence or absence of the signal is important.

### 3.2. *SWSL*

SWSL provides language constructs to represent the components of the workload model in order to specify SWs.

A SW is not a real application. It does not execute on real data; it merely produces the same workload characteristics as an application which does execute on real data. Since there are no real data values upon which to base branching decisions, branching, and thus looping, are done probabilistically. Similarly, in a real application, some behaviors of the workload occur in response to random external events. To model this behavior, the invocation time of sporadic tasks and the generation times for asynchronous data from terminators are determined probabilistically. SWSL provides random number generators with a range of probability distributions (e.g., uniform, geometric, Poisson, etc.) for use in specifying loop counts, task deadlines, invocation times for sporadic transformations, and rates for terminators. Probabilistic branching and loop counts were also used in the SW specification language for Pegasus (Singh and Segall, 1982).

Likewise, real-time systems control physical processes. For some applications, the system cannot be evaluated while connected to live sensors and actuators. This is especially true when using a SW, which cannot properly interpret data from sensors nor send the proper commands and data to actuators. For this reason, the SW simulates the input and output behavior of the workload via synthetic terminators. Our implementation of synthetic terminators is an expansion of external event generation (Singh and Segall, 1982) and techniques for device simulation, e.g., (Gomaa, 1986).

A SWSL specification consists of three files. The first file contains the specification of the dataflow graph using the components described above. It also contains the assignment of each component to its appropriate processor in the distributed system. The second file contains the specification for the functions executed by the transformations. Synthetic operations are defined to produce the effects of computation, data movement, and I/O. Loops are specified with a fixed or randomly generated loop count. In branches, probabilities are specified for each of the possible branch paths.

The third file contains parameters used by the SW to control specific behaviors in an experimental context. These include the duration of the experimental run and seeds for the various random number generator streams used by the SW to simulate stochastic behavior. The use of these parameters is described in the Section 4.

### 3.3. *Synthetic Workload Generation*

Our SWG compiles SWSL specifications and thus completely automates the generation of SWs. The synthetic workload generation process is shown in Figure 3. 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. Next, it compiles the experiment

file. Then, it compiles the functions file and produces C language code for each function. 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 uses the processor assignment information from the graph file to direct the *make* utility.

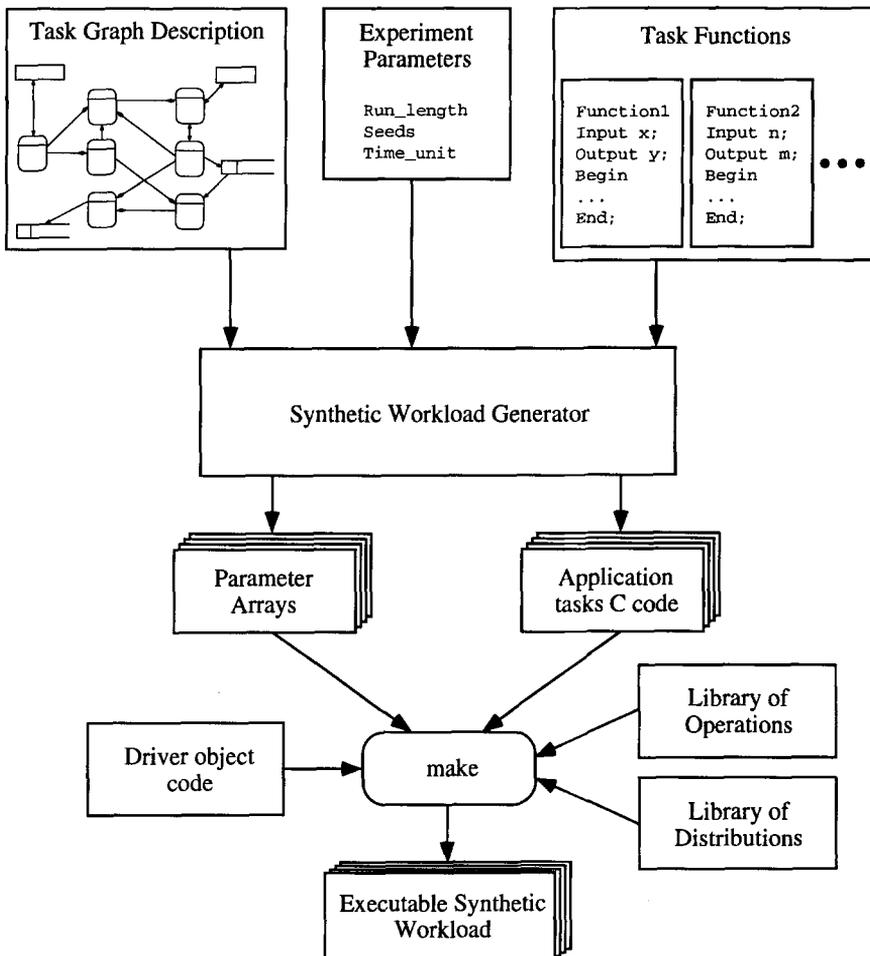


Figure 3. Synthetic Workload Generation.

## 4. Experimentation Using a Synthetic Workload

The purpose of the SW is to aid in performing experiments on HARTS. The SW code is developed and compiled on a separate workstation and the executable code downloaded to the various processors in HARTS, via the Ethernet. An experiment generally consists of the following steps.

1. Specify the SW using SWSL.
2. Compile the SW.
3. Download the executable code to the HARTS processors.
4. Execute the SW.
5. Collect performance data.

These steps are then repeated for each subsequent experiment. To minimize the time between experiments, a multiple run feature has been implemented in the SW. With this feature, the parameters for a number of experiments are specified at one time. Then the SW is compiled and downloaded. Next, steps 4 and 5 may be repeated to perform a number of consecutive experiments. After the execution of each run, the SW waits for input from the processor's console. This wait gives the user time to upload performance data or reset measurement instruments before the system is reinitialized for the next run.

The length of a run can be specified by the `TIMEOUT` parameter in the experiment file. To ensure statistical independence of performance measurements, no history information is kept between runs. All tasks and data structures are deleted at the end of each run and reinitialized at the beginning of the next run. Also, all random number generator streams are reinitialized as specified by the user.

## 5. Synthetic Workload Structure

### 5.1. Overview

The synthetic workload (SW) executes on a distributed real-time system. Our target system, HARTS, does not have shared memory, nor does it support remote invocation of tasks. Therefore, distributed control must be used in the SW. This control strategy requires that the SW on each processor be composed of two groups of processes: the *synthetic application tasks*<sup>1</sup> and the *driver processes*. The synthetic application tasks implement the user-specified SW. The driver controls the SW in the context of an experiment.

The graphical representation of the SW for each processor is shown in Figure 4. The driver processes and data structures are shown in the labeled box. All other transformations, stores, and terminators represent the user specified SW.

The use of a driver to control a SW is not a new concept. A good discussion on workload drivers may be found in (Ferrari, 1978). Our contribution is to describe the

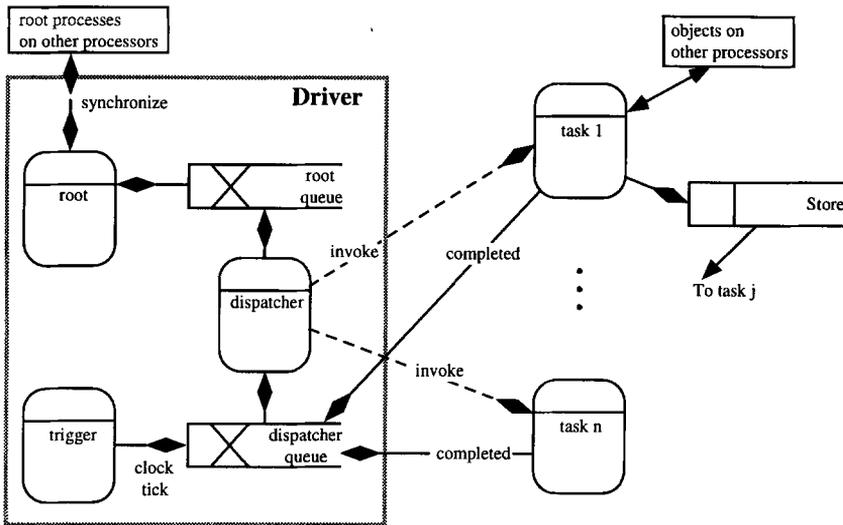


Figure 4. Data flow model for the SW processes on a single processor.

important features of a driver for SWs to be used in distributed real-time systems. These features have been implemented in the driver for our SW.

Our distributed control mechanism is designed to minimize communication overhead which would adversely affect the performance indices being measured. The only communication overhead is caused when the distributed SW drivers synchronize at the beginning and end of each run. Synchronization is necessary if synthetic tasks on different processors require synchronous communication. It is especially important if the tasks are periodic. Synchronous communication between unsynchronized, periodic tasks on different processors can cause intolerably long waits for the task which is invoked first as it waits for the other task to be invoked. The SW is different from a distributed discrete event simulation, where simulation tasks execute in simulated time. If some processors start later than others, then the algorithms which maintain global consistency of simulation time can quickly bring the processors into relative synchrony. In the SW, all tasks run in real ("wall clock") time. The execution cycles of periodic tasks are fixed and are independent of the behavior of the periodic tasks. They will not become synchronized through the actions of the tasks. Explicit synchronization is required.

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

In addition to these behavioral features, the SW has implementation features to make it easier to use and easier to port to a new system. Although the synthetic tasks may change

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

### 5.2. *Synthetic Application Tasks*

The synthetic application tasks are responsible for generating the resource demands on the processor. They are the implementations of the transformations in the SWSL specification. They execute code generated from the function file. Computation and communication are performed using synthetic operations. We have developed a library of synthetic operations. A synthetic operation may be called from a SWSL function to produce a specific resource demand or communication behavior. Examples of these operations are `float(n)` and `write(d)`. The former performs  $n$  floating point arithmetic operations, and the latter sends a message to the destination exchange  $d$ . The size of the message is determined by the size of the data elements specified for  $d$ . The probabilistic branching and looping constructs use independent random number generator streams. A library of random number generators for different distribution functions is available.

### 5.3. *Driver Processes*

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

The *root* process executes first at system initialization. It spawns all the other driver processes and synthetic application tasks and creates the data structures that implement the stores. It also synchronizes with the root processes on the other processors at the beginning and end of each execution.

The driver uses the `TIMEOUT` parameter from the experiment file to specify the maximum time that each run is to execute. When this time is reached, the run ends. At the end of a run, all SW tasks, data structures, and processes, except for the root process, are deleted. No history information is kept between runs. Hence, runs are statistically independent. After the execution of each run, the root process waits for input from the user. This wait gives the user time to upload performance data or reset measurement instruments before the system is reinitialized for the next run.

The two other driver processes are the *trigger* and *dispatcher*. Together, they provide a facility for dispatching periodic and asynchronous tasks to the pSOS scheduler, which does not directly support periodic scheduling. The trigger acts as a software timer that periodically sends clock tick messages to the dispatcher. The time interval between these messages is specified with the `TIME_UNIT` parameter in the experiment file.

The dispatcher uses the trigger messages to count time. The time value is used for the scheduling of activities, which, on HARTS, include dispatching periodic tasks and the enforcement of task deadlines. On a system whose operating system fully supported periodic tasks, these functions of the dispatcher would not be necessary. Therefore, scheduling of periodic tasks and enforcement of deadlines are not considered to be permanent parts of the SW. The dispatcher maintains an activity<sup>2</sup> queue which is similar to event queues used in discrete event simulation systems. Activities correspond to actions that are to be performed by the dispatcher at specific times. Activities indicate each task's start time, as defined by the `START_TIME`, `PERIOD`, and/or `SPORADIC` parameters, and deadline, as specified by the `DEADLINE` parameter. There are also activities to indicate the times when the simulated terminators are to produce or consume data.

The dispatcher uses the `RATE` parameter for terminators to determine when to send messages or events from source terminators and when to read messages or events at sink terminators. The SW simulates terminators by using a data structure and a task for each terminator. The data structure is used if the simulated terminator is to produce or receive data. Synthetic application tasks that communicate with the terminator will send and receive messages to/from the data structure. The terminator task is used if the simulated terminator is to produce or receive signalled events, since only tasks may send and receive events in pSOS. When instructed by the dispatcher, it sends signals to the appropriate application tasks or reads signals sent by application tasks.

When the dispatcher invokes each task, if that task has a specified deadline, then the dispatcher places a deadline activity with the proper time value on the activity queue. Each task sends a message to the dispatcher when it has completed execution. (It is assumed that a task executes its function completely within each period.) The dispatcher uses this information to cancel the corresponding task deadline activity in the activity queue. If the time indicated for a deadline activity is ever reached, then the dispatcher kills the corresponding task and creates another task with the same characteristics. This ensures that the task will be in its initial state when it is next invoked.

## 6. Driver Overhead

As was stated earlier, the application tasks are to produce the desired workload characteristics. It is their structure and behavior which has been specified by the user. The structure of the driver is fixed. Obviously, its behavior is influenced by the workload parameters. The dispatcher, for example, will execute more frequently in a workload with short period tasks than it will in a workload with longer period tasks. However, the amount of time that it executes per task invocation will be fixed. The driver thus produces a calculable overhead per task execution. This overhead may be measured and taken into account as the workload is being tuned for a particular experiment. Since we are working with real-time systems, we want the driver overhead to be as low as possible. We want to minimize the amount that the driver perturbs the performance of the application tasks.

We ran the synthetic workload on HARTS in order to measure the driver overhead. The workload consisted of four application tasks. The execution of each task consisted of

simply a loop to use CPU time. Since the per invocation overhead is fixed, this workload was sufficient. Data was collected for each of the three driver processes (root, dispatcher, and trigger). The dispatcher time was measured separately for each of the functions that it performs. These functions include: invoking a task, aborting a task which has reached its deadline, and performing a time update. The times given for each operation is the time from when the dispatcher reads the message until immediately before it makes the system call to request the next message. The time to perform this system call is of the order of 100  $\mu$ sec. The average time for a single task invocation was 118  $\mu$ sec. The time here is clearly dominated by the approximately 100  $\mu$ sec. it takes to perform the `resume.p()` system call which is used to invoke a task. The time to process a deadline event is 469  $\mu$ sec. This time is inflated by the number of system calls that must be performed to destroy the task and spawn and activate a new version of it.

The trigger process took 247  $\mu$ sec. per trigger. The actual amount of code in the trigger process is small, but the execution time is inflated because two pSOS system calls are made per trigger. Finally, the root process executed for less than 60 milliseconds total. This time is primarily used in system initialization. The root process is suspended while the rest of the workload is executing.

## 7. Summary and Conclusions

A SW is a useful tool for use in performance analysis of prototype systems. It allows the user to create arbitrary workloads. These workloads may be representative of real workloads, or they may be designed to represent artificially extreme operating conditions. This capability is achieved by allowing full flexibility in the SW application tasks in addition to the parameterized SWSL specification. Experiments demonstrating both the ability of the SW to be representative of a real robot control system and the ability to generate specific, controlled workload characteristics are described in (Kiskis, 1992).

By creating SWSL, the SWG, and the SW, we have provided a tool for generating SWs. The SWG does not have to be used as a stand-alone tool. Because we have based the workload model, and hence, SWSL, on ESML, an established software specification notation, the SWG can be integrated into toolsets which support ESML or similar notations. In such a toolset, the high-level software specification in ESML could be augmented with SWSL parameters. Such a system would allow the developers to produce an SW which is representative of the software they are building. Of course, the level of representativeness is determined by the ability of the developers to accurately predict the resource demands of the tasks being modeled. As the various software components are developed, the real code may be used in place of the corresponding components of the SW. The caveat here is that, as described in Section 3.2, the synthetic tasks cannot manipulate real data and, thus, cannot interact with the real tasks in complex ways. Therefore, the level of granularity for replacement of SW components with real code may be relatively coarse.

The primary enhancement planned for the SW is to enable the specification and generation of SWs for Posix compliant systems. This will require an evaluation of all components of the synthetic workload generation system, from workload model to SW,

to determine what modifications must be made to support the wide range of tasking, resource usage, and interprocess communication that is supported in such systems. Due to the generality of the workload model, these changes should be isolated to the selection of object parameters and the implementation of the driver and workload objects.

## Notes

1. The term "task" is used to distinguish the synthetic application processes from the driver processes.
2. The term "activity" is used to avoid confusion with events in the workload model.

## References

- W. Bruyn, R. Jensen, D. Keskar, and P. Ward. ESML: An extended systems modeling language based on the data flow diagram. *ACM Software Engineering Notes*, 13(1):58–67, 1988.
- W. Buchholz. A synthetic job for measuring system performance. *IBM Systems Journal*, 8(4):309–318, 1969.
- F. Feather. Validation of a fault-tolerant multiprocessor: Baseline experiments and workload implementation. Master's thesis, ECE Dept., Carnegie-Mellon University, Pittsburgh, 1984.
- F. Feather, D. Siewiorek, and Z. Segall. Validation of a fault-tolerant multiprocessor: Synthetic workload implementation. In *Proc. Int'l Conf. on Distributed Computing Systems*, pages 303–312, May 1986.
- D. Ferrari. *Computer Systems Performance Evaluation*. Prentice-Hall, Englewood Cliffs, 1978.
- D. Ferrari. A performance-oriented procedure for modeling interactive workloads. In D. Ferrari and M. Spadoni, editors, *Experimental Computer Performance Evaluation*, pages 57–78, New York, 1981. North-Holland.
- D. Ferrari. On the foundations of artificial workload design. In *Proc. of 1984 ACM SIGMETRICS Conf. on Meas. and Modeling of Comp. Sys.*, pages 8–14, August 1984.
- H. Gomaa. Software development of real-time systems. *Communications of the ACM*, 29(7):657–668, July 1986.
- Daniel L. Kiskis. *Generation of Synthetic Workloads for Distributed Real-Time Computing Systems*. PhD thesis, University of Michigan, August 1992.
- Daniel L. Kiskis and Kang G. Shin. SWSL: A synthetic workload specification language for real-time systems. *IEEE Trans. Software Engineering*, 20(10):798–811, oct 1994.
- H. F. Ledgard and M. Marcotty. A genealogy of control structures. *Communications of the ACM*, 18(11):629–639, November 1975.
- H. C. Lucas, Jr. Synthetic program specifications for performance evaluation. In *Proc. ACM Annual Conference*, pages 1041–1058, Boston, August 1972.
- H. D. Schwetman and J. C. Brown. An experimental study of computer system performance. In *Proc. ACM Annual Conference*, pages 693–703, 1972.
- K. G. Shin, D. D. Kandlur, D. L. Kiskis, P. S. Dodd, H. A. Rosenberg, and A. Indiresan. A distributed real-time operating system. *IEEE Software*, pages 58–68, September 1992.
- Kang G. Shin. HARTS: A distributed real-time architecture. *IEEE Computer*, 24(5):25–35, May 1991.
- A. Singh and Z. Segall. Synthetic workload generation for experimentation with multiprocessors. In *Proc. Int'l Conf. on Distributed Computing Systems*, pages 778–785, 1982.
- Software Components Group, Santa Clara, CA. *pSOS User's Guide*, 1986.
- K. Sreenivasan and A. J. Kleinman. On the construction of a representative synthetic workload. *Communications of the ACM*, 17(3):127–133, March 1974.
- H. Tokuda and M. Kotera. Scheduler 1-2-3: An interactive schedulability analyzer for real-time systems. In *Proc. of the 12th Annual Int'l Computer Software & Applications Conference*, pages 211–219, 1988.
- D. C. Wood and E. H. Forman. Throughput measurement using a synthetic job stream. In *AFIPS Fall Joint Computer Conference*, volume 39, pages 51–55, November 1971.