Real-Time Systems, 4, 287–306 (1992)
© 1992 Kluwer Academic Publishers. Manufactured in The Netherlands.

Modeling and Evaluating the Feasibility of Timing Constraints Under Different Real-Time Scheduling Algorithms

STEVEN BERRYMAN AND IAN SOMMERVILLE*
Computing Department, Lancaster University, Lancaster, LAI 4YR UK

Abstract. The objective of the work described here is to provide a software tool to assist real-time system specifiers and designers to predict, at an early stage of the development process, the timing behavior of the system being developed. The timing behavior of the system is dependent on the scheduler which is the central component of any real-time system. Our tool (Simulation of Real-Time systems (SRT)) is used to model the timing aspects of a real-time system and to simulate the system with a particular scheduling policy so as to predict its behavior. This paper will present an overview of the capabilities of the system which is implemented using an object-oriented programming language (Smalltalk-80).

1. Introduction

A real-time system is expected to interact with the environment within certain timing constraints and software designers must produce a system which can guarantee to meet these constraints. A realistic real-time system is composed of many interacting modules which can be executed with real or virtual concurrency. Real concurrency describes the situation in which each module executes on its own processor whereas virtual concurrency is that in which many modules (processes) are executed on one processor (Allworth 1981). There can also be a combination of these, where many processes are distributed between many processors. Even the smallest real-time systems can, therefore, present considerable timing problems which may be addressed using systematic methods supported by software tools (Orr 1988). It is particularly important that software tools are available to help system designers during the design rather than implementation at which stage finding a fault would be far more costly (Malcolm 1989).

There are many factors which influence the timing behavior of a real-time system and any alteration of the system can radically change its behavior. For example, the effect of an additional requirement in one process could propagate through the system and cause a failure in another process. It may not be intuitively obvious which process(es) would fail since this would depend on the scheduling policy implemented. Any alterations in the timing behavior will alter the way in which the scheduler policy manages its resources; thus the scheduling policy is the central influencing factor on the timing behavior of the system. In order for a tool to predict the timing behavior of a real-time system, it must take into account the scheduling policy to be used.

*This work has been jointly funded by the Science and Engineering Research Council, UK and British Aerospace, Warton, UK.

Our tool SRT (Simulation of Real-Time systems) (Berryman 1991), allows a model of a real-time system to be constructed and then evaluated, by simulation, using a particular scheduling policy. The construction of the model is achieved by using a graphical user interface (Figure 1). Icons from the control panel are copied onto the design panel and then joined with lines that represent databuses. Each icon has a number of attributes which can be specified by the designer of the simulation.

The SRT model is only concerned with the timing aspects of real-time systems and does not address the prototyping of functionality. The reasons for this design decision are:

- It saves time in constructing expensive prototypes to find out the same timing information. Also it is difficult to verify timing constraints using prototypes since speed and efficiency are relaxed so as to reduce the development time of the prototype (Sommerville 1992).
- SRT can evaluate the basic design before any detailed functional design is carried out. This avoids wasting time on designing a system for which the requirements are unfeasible.

The system designer can therefore use SRT to model the timing behavior of a real-time system before significant implementation effort is devoted to the system functions.

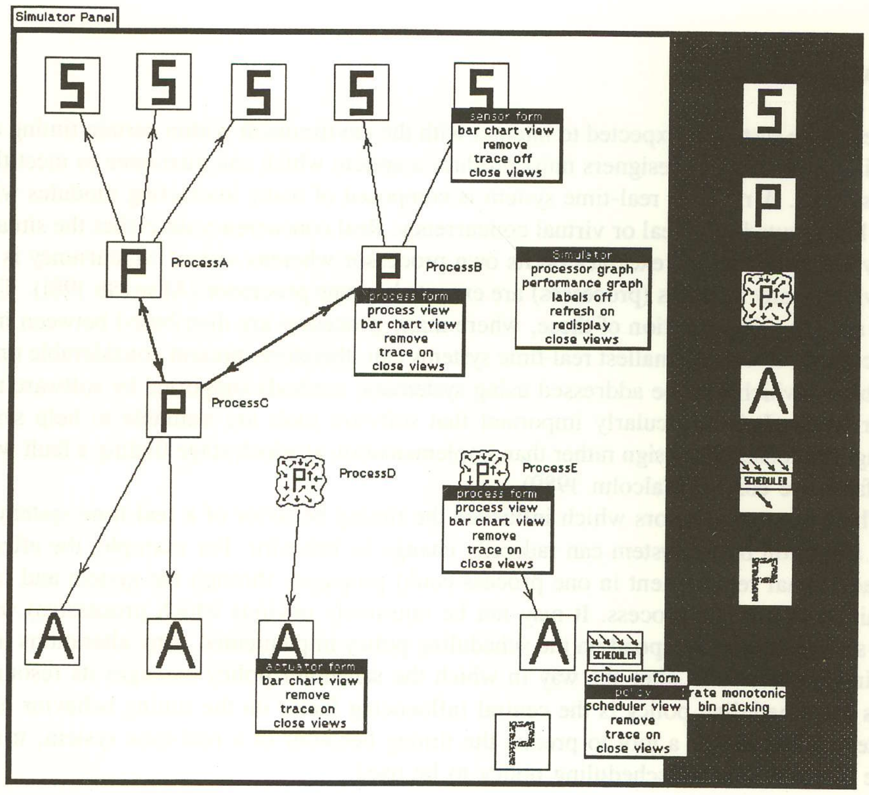
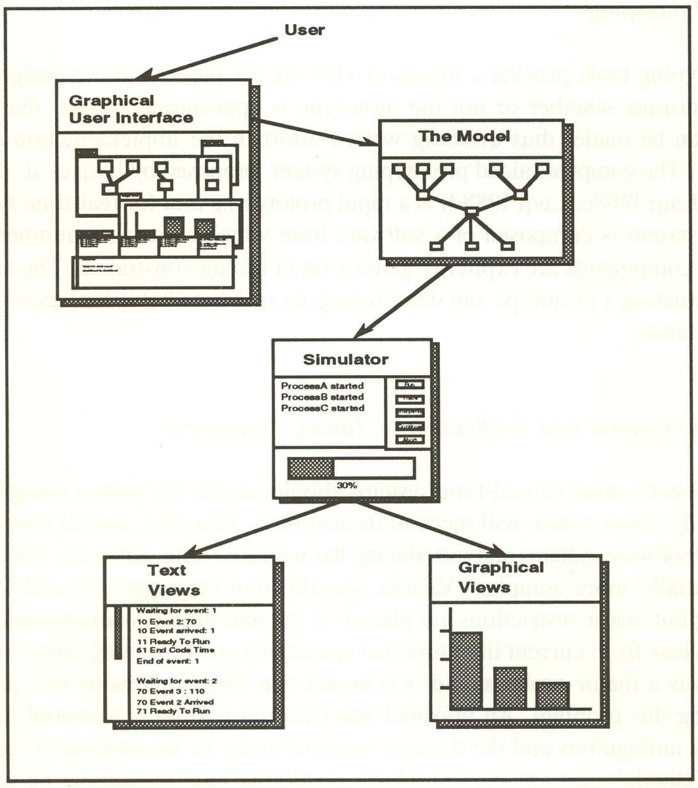


Figure 1. Simulation panel.



```
ProcessA started
ProcessB started
ProcessC started
```

```
Waiting for event: 1
10 Event 2: 70
10 Event arrived: 1
11 Ready To Run
61 End Code Time
End of event: 1

Waiting for event: 2
70 Event 3 : 110
70 Event 2 Arrived
71 Ready To Run
```

Figure 2. SRT's architecture.

This paper presents an overview of the system which is implemented using Smalltalk-80, an object-oriented programming language. The language was chosen because of its rapid prototyping ability, thus allowing ideas to be implemented and tested very quickly. Figure 2 shows an overview of SRT's architecture.

This paper is organized as follows. In Section 2, we look at current tools and method for modeling timing constraints. In Section 3, we describe how a real-time system can be modeled using SRT. In Section 4, we look at two real-time scheduling algorithms which have been incorporated into SRT. In Section 5, we look at how the model can be evaluated using various scheduling policies and then analyze the output data produced. In Section 6, we look at some ideas for future versions of the tool and draw some conclusions about our work.

2. Related Work

There are various tools and methods for prototyping, formally specifying and implementing real-time systems. This section looks briefly at each area and concludes by placing SRT in context.

2.1. *Rapid Prototyping*

Rapid prototyping tools provide a means of viewing the product at the design stage. The user can determine whether or not the prototype is appropriate. If not, the appropriate alterations can be made, thus avoiding wasted effort in the implementation of incorrect requirements. The computer aided prototyping system, proposed by Luqi et al. (Luqi 1988a, Luqi 1988b, Luqi 1988c, Luqi 1988d) is a rapid prototyping tool for real-time system prototyping. The system is composed of a software base which contains a number of components. These components are explicitly given a set of timing constraints. The tools provide help in constructing a prototype and determining its real-time behavior based on the given timing constraints.

2.2. *The Specification and Verification of Timing Constraints*

The formal specification of real-time systems should enable the system designer to verify mathematically that a system will meet all its deadlines. However, formal methods are still immature for software systems and introducing the notion of time makes the issue of verification exponentially more complex. Various specification languages for real-time systems are emerging but major restrictions are placed on the real-time systems to reduce the complexity. It is clear from current literature that specifying and designing large complex real-time systems is a major problem and it is hoped that formal methods will go some way toward solving this problem. An informal specification written in a natural language can contain many ambiguities and the timing constraints may be incomplete or inconsistently specified. Methodologies are needed which enable the requirements to be captured and specified in such a way that inconsistencies within the specification can be eliminated.

ESTEREL (Berry 1983) was one of the first languages to emerge to support the specification of real-time systems. The language enables time to be expressed and verified within the specification. Dasarathy (Dasarathy 1985) went on to investigate the various timing constraints that would be required in a specification language and classified them into two categories, namely performance and behavioral constraints. The performance constraints define the response time of the proposed real-time system and the behavioral constraints define the rate at which external stimuli are applied to the system. This classification has lead to the development of timing constructs in many of the existing formal models.

The finite state model has been one of the most widely used models for the development of specification languages; examples of such languages are RT-ASLAN (Auernheimer 1986), Statecharts (Harel 1988) and Modecharts (Jahanian 1988). RT-ASLAN enables the designer to specify the system and its behavioral constraints in a formal notation which can then be used to verify certain functional and behavioral aspects of the system. This specification language makes the simplistic assumption that each process has its own dedicated processor, thus avoiding any scheduling issues.

Statecharts and Modecharts are graphical specification languages. Statecharts are based on the extended finite state machine model with the timing constraints being expressed in temporal logic. The Modechart language is an extension to Statecharts in that the timing constraints have been explicitly included into the graphical model. Methods have been developed to reason about the timing properties of the system and try to verify the timing constraints (Jahanian 1989, Jahanian 1988). Ideas were also proposed for a Modechart simulator (Stuart 1991) which enables the designer to specify the distribution of events and to simulate the execution in order to predict the timing behavior of the system.

The approach described in Lauber's paper (Lauber 1989) is based on the assumption that the design of large complex real-time systems will use a specification language. If timing constraints can be expressed within the specification then it can be used to test the behavior of the real-time system. The specification model represents a rough prototype which can be evaluated. The methods used are based on dynamic testing of the specification with data being fed in either by the systems designer or by a simulation of the real world.

2.3. *Real-Time Programming Languages*

Real-time languages have been developed to enable programmers to express timing constraints within the language. Only when the time dimension is incorporated explicitly into the language can time constrained computation be expressed adequately and conveniently (Halang 1990). The real-time language Euclid (Kligerman 1986) is capable of expressing timing constraints but at the cost of restricting other unpredictable features such as dynamic process creation and recursion. Real-Time Concurrent C (Gehani 1991) offers more flexibility than Euclid by enabling the programmer to use dynamic constructs but time and space bounds are required. The language Flex (Lin 1988) supports facilities for imprecise computing, which is a method of arriving at an approximate value and then improving the result as much as time permits. If a task's resources are not available, the scheduler can dynamically alter the execution time of the task so that it can be completed before its deadline.

These languages offer different levels of flexibility. Euclid requires all timing constraints to be static and computed at compile time. Real-Time Concurrent C facilitates the use of dynamic constructs as long as the space and time bounds can be computed at compile-time. The Flex language does not require the systems resources to be extensively over-specified because the program can dynamically change itself to fit around the current available resources.

2.3.1. Why the SRT Model is Necessary. All the approaches described above have a particular place in the software development cycle. The prototyping approach concentrates on modeling the requirements. To build a complete prototype the design would need to be decomposed into modules and then these modules implemented in the prototyping language. Thus, to determine the feasibility of the timing constraints from this method requires a complete breakdown of the design. A few prototyping/specification languages, such as PAISLey (Zave 1986), can tolerate an incomplete prototype for execution but their objectives are to evaluate the functional requirements, thus ignoring the behavioral requirements.

Current formal methods are not mature enough to represent the complexities of realistic real-time systems. The Modechart approach has shown much promise but mechanisms are required by the designer which provide information about the behavior of the system without building a complete functional model. Many of the formal specification methods are still very difficult to use for the nonspecialist and very cumbersome, if not impossible, to use with large complex real-time systems. Lauber's method involves a detailed specification, much of which is irrelevant when dealing with timing issues. This method also makes the assumption that all systems are formally specified whereas in reality most are not.

Real-time languages offer the programmer the facility to express and evaluate timing constraints but determining the feasibility of these constraints is necessary well before implementation.

Tools are required which enable the designer to rapidly build a skeleton prototype of a real-time system and then evaluate this prototype to determine information about the system's behavior. All of these methods are concerned with modeling both the functionality and timing aspects of the system, whereas our work deals with the timing aspects alone.

3. Modeling Timing Constraints

Real-time systems can be considered, at an abstract level, to consist of three basic entities, namely sensors, processes and actuators. SRT uses these *real-time entities* to model the process structure and proposed hardware devices. The process structure is made up of a fixed number of processes, because dynamic process creation causes unpredictability. The real-time language Euclid (Kligerman 1986) has made the same assumption and the authors claim that no functionality is lost, due to the fact that most real-time applications tend to consist of a fixed number of controllable resources.

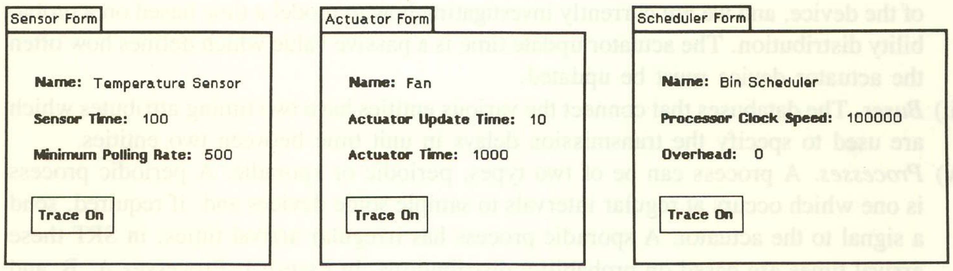
The time constraints, which the software designers must adhere to, will be defined by the system designers. These timing constraints will be determined by the environment, so that the systems designer has to calculate how quickly the system must respond to changes in the environment.

Figure 1 shows five Processes; ProcessesA and B are polling a number of sensors which then synchronize with ProcessC. ProcessC waits for data from both sensors and then activates the actuators. ProcessD and ProcessE are waiting for some external event which, on arrival, will activate the actuator. All processes on this panel are on the same processor and the scheduler determines which process will be activated next. The background process will be executed if no other process needs to be run.

3.1. Modeling a Real-time System using SRT

SRT consists of five RT entities, which are represented as icons on the simulation control panel (Figure 1). Sensors, processes and actuators are used as the basis for constructing the real-time prototype. Two other entities, scheduler and background are special types of process that must always be present. Entities are connected by databuses, which are represented as arrows on the simulation control panel. Each entity has a number of attributes whose values the designer estimates then enters by filling in forms. Figure 3 shows examples of such forms.

3.1.1. Timing Attributes. Each of the three Real-Time entities; sensors, processes and actuators, have attributes which define the timing requirements of the prototype. They are of two types:


Sensor Form

Name: Temperature Sensor

Sensor Time: 100

Minimum Polling Rate: 500

Trace On

Actuator Form

Name: Fan

Actuator Update Time: 10

Actuator Time: 1000

Trace On

Scheduler Form

Name: Bin Scheduler

Processor Clock Speed: 100000

Overhead: 0

Trace On

Figure 3. Entry forms.

i) *Actual Time.* This timing attribute represents the actual measurement of time in the real world which is typically used to simulate a delay for an I/O device. The actual time in SRT is specified in microseconds.
ii) *Unit Time.* This is the number of units (clock cycles) a process requires from the processor to simulate the execution of a piece of code. This type of timing is processor dependent, so that changing the processor speed will result in the unit time taking different values of actual time.

3.1.2. Real-Time Entities. This section describes four of the RT entities (sensors, processes, actuators and background process) and the databus relationship. The scheduler entity will be explained in detail in Section 4.

i) *Sensors.* The sensor device receives a signal from a process and then replies with its current state. The sensor has two timing attributes, sensor time and minimum polling rate. The sensor time attribute is the actual time in microseconds that a sensor takes to receive a signal from the bus, convert the sensor reading from analog to digital and put the result back onto the bus. The second attribute is a constraint on the sensor device which defines how often the device should be polled for the system to meet its response time. This attribute is a passive value as it is not used by the simulation process but is used in the output to determine whether the system's behavior has met its timing requirements.
ii) *Actuators.* A process can trigger an actuator in one of two modes, synchronous or asynchronous. In synchronous mode, the process sends a signal to the actuator and waits for a reply (suspends) before it continues. The asynchronous mode does not wait for a reply from the actuator but signals to the actuator and then continues. This mode may result in a queue building up on the actuator if the process puts out more signals than the actuator can consume. The importance of this varies with the device. For example, if the actuator is a graphical display, then losing a few signals may be acceptable but in other situations an overload could be disastrous.

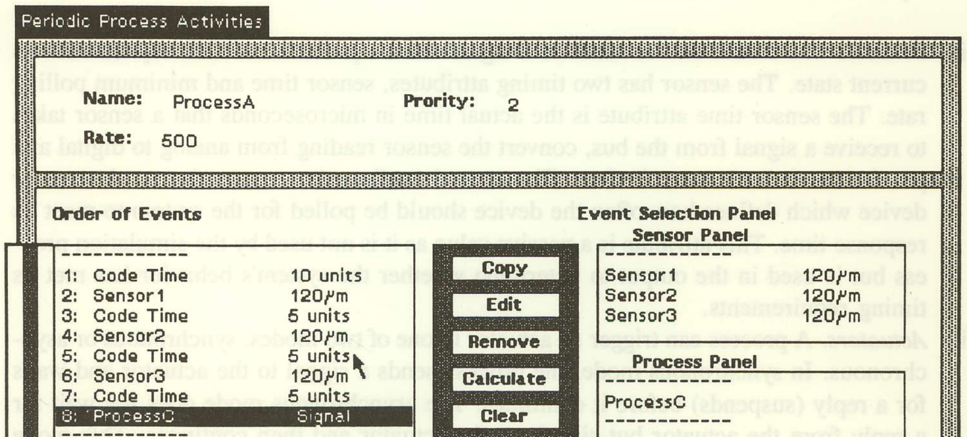
    The actuator has two timing attributes, actuator time and actuator update time. Actuator time is the actual time it will take the device to complete its task. Currently this time is static, so that the actuator always takes a set amount of time. Realistically, the amount of time taken to complete the task would vary according to the current state of the device, and we are currently investigating how to model a time based on a probability distribution. The actuator update time is a passive value which defines how often the actuator device must be updated.

iii) *Buses.* The databuses that connect the various entities have two timing attributes which are used to specify the transmission delays in unit time between two entities.

iv) *Processes.* A process can be of two types, periodic or sporadic. A periodic process is one which occurs at regular intervals to sample some devices and, if required, send a signal to the actuator. A sporadic process has irregular arrival times; in SRT these arrival times are based on probability distributions. In Figure 1, Processes A, B, and C are icons for periodic processes whereas Processes D and E are icons for sporadic processes.

A process can poll sensors, do some processing, synchronize with other processes and activate actuators. Memory can be modeled as a separate process with which other processes have to synchronize before they can continue. A process may suspend for any of three reasons: i) to wait for data from a sensor; ii) to wait on a semaphore; iii) to wait for the completion of an actuator in synchronous mode. During the execution of code it is possible for another process with a higher priority to pre-empt it.

Figure 4 is a process activity view of a periodic process, which enables the software designer to define the process's characteristics. The top panel contains the process's attributes. The rate attribute defines how often this process should be executed each second and the priority attribute determines the importance of the process. A sporadic process



```
Name:  ProcessA                Prority:  2
Rate:  500

Order of Events                          Event Selection Panel
                                         Sensor Panel
-------------                            ------------
1: Code Time        10 units             Sensor1     120/m
2: Sensor1          120/m                Sensor2     120/m
3: Code Time        5 units              Sensor3     120/m
4: Sensor2          120/m
5: Code Time        5 units              Process Panel
6: Sensor3          120/m                ProcessC
7: Code Time        5 units              ------------
8: ProcessC         Signal
                                         Actuator Panel
Total Time:  610/m
Processor Units: 61
Approx:  1639 times per second
```

Figure 4. Process activity list.

has different attributes. Instead of a rate attribute it has a probability distribution, an earliest possible start time and a deadline. The middle panel contains a number of selection lists which are used to describe the process's minor cycles. Figure 4 shows that processA first of all executes some code to start up, then sensorsl, 2 and 3 are polled with a small amount of code executed in between them. Finally it sends a signal to ProcessC's semaphore.

The information on the bottom of the panel is calculated from one major cycle of the process. One major cycle is defined as being the complete execution of all the steps in the process activity list as shown in Figure 4. A minor cycle is defined as being one of the steps in the process activity list. The figures show the amount of actual time a process will take, the number of processor units required and finally the number of times this process could feasibly be executed per second if it had sole access to the processor. Clearly this figure should be significantly greater than the rate specified at the top of the panel. These are *best-case* figures as they assume no other process will preempt it.

v) *Background process.* This process is executed only when nothing else can be run. Generally this type of process in real-time systems will carry out checks on the hardware. No timing attributes are associated with this process.

3.2. Incorporating Timing Constraints into the Model

SRT has two types of timing constraints, process deadlines and device deadlines.

i) *Process Deadlines*
The process deadline is defined in two different ways depending on the process type (sporadic or periodic).
- Periodic Processes. Each process has a frequency rate which implicitly defines its deadline. The process must be completed before it is next scheduled otherwise an overload will occur. An overload will be dealt with in different ways depending on the scheduling algorithms currently employed.
- Sporadic Processes. When an event arrives, the process must start execution after the earliest start time and finish before the deadline. These parameters are defined by the designer.

ii) *Device Deadlines*
These are constraints placed on the system by the system designer when determining the sampling and action rate for the system to function within the requirements. The computer system must be able to meet these constraints for the system to function correctly. For example, a display may have a device constraint stating that it must be refreshed 30 times a second.

4. The Scheduler

SRT Currently incorporates two scheduling algorithms, namely the rate monotonic algorithm (Liu 1973) and the bin packing algorithm (Blake 1991). The rate monotonic algorithm is a fixed priority scheduling algorithm, whereas the other algorithm is a dynamic priority scheduling algorithm. The next two sections will briefly explain the above algorithms, a more detailed explanation can be found in the above references.

4.1. *Rate Monotonic Scheduler*

This scheduling algorithm decides what process to run by considering the priority and state of the process. A process can be one of the following states: running, ready, suspended or dormant as shown in Figure 5. The running process is the one currently being executed on the processor. A process will be in the ready state when it is ready to be executed but a higher priority process is currently running. The suspended state is entered while the process is waiting for a device to finish or waiting for another process to synchronize with it. The dormant state is when a process has either finished its major cycle and is waiting for its next occurrence or the process has failed to meet its deadline. A process in the dormant state will move to the ready state on its next occurrence.

4.2. *Bin-Packing Scheduler*

This algorithm is a dynamic priority scheduling algorithm in which processes join an arrival queue which the scheduler then tries to schedule so that each process makes its deadline, see Figure 6. It is assumed that process execution times, deadlines and earliest possible start times are known in advance.

The bin-packing scheduler looks ahead and will only schedule a process if its timing constraints can be guaranteed to be met. A process is scheduled to the nearest point before its deadline. If it is not possible to fit the process between the start time and deadline then

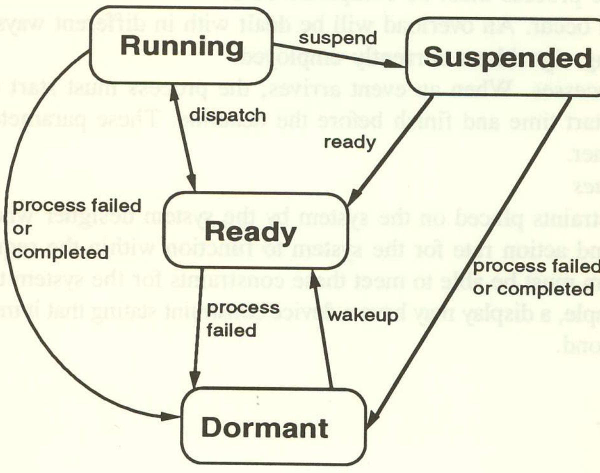


Figure 5. The various states of a process.

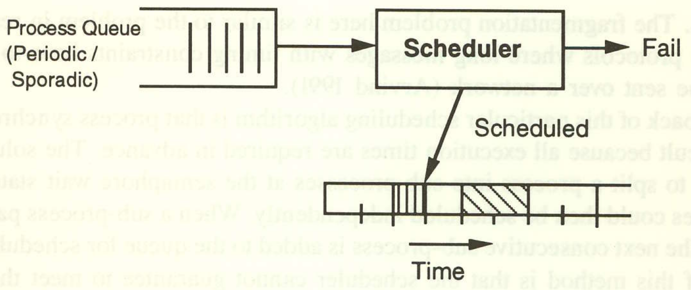


Figure 6. Process queue.

the process will fail. A process can be broken into smaller pieces so as to fit it in within the bounds. Figure 7 shows a small example of two processes being scheduled using this algorithm.

ProcessA	—	A periodic process which occurs every 40 clock cycles and has an execution time of 10.
ProcessB	—	A sporadic process which arrived at time 5, start time of 20 (actual start time = 20 + 5), execution time of 15 and a deadline of 45 (actual deadline = 45 + 5).

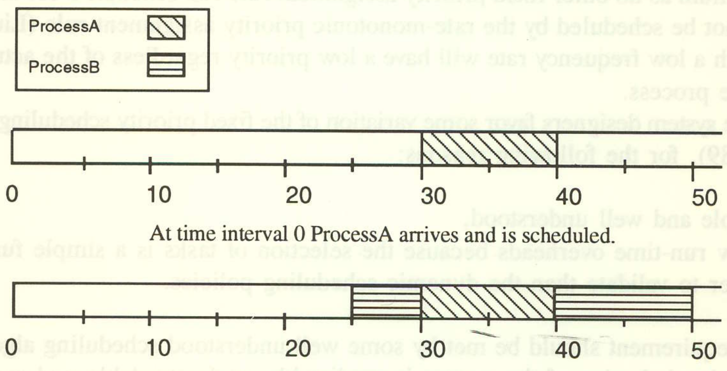


Figure 7. Bin packing scheduler.

A process in SRT (and any realistic real-time system) is composed of a number of minor cycles which do not have individual deadlines but rather a deadline for the whole major cycle. For the above algorithm to be implemented under the SRT model, an intermediate layer is required which divides up a process into separate executable pieces which can then be sent to the bin-packing scheduler. The intermediate layer is also responsible for propagating the overall process deadline through each individual executable piece of code. This propagation of deadlines will take into account all the I/O transactions within the process; this presents no problem as I/O response times will be known when the process arrives at the queue. The fragmentation problem here is similar to the problem in real-time communications protocols where long messages with timing constraints have to be split into packets to be sent over a network (Arvind 1991).

The drawback of this particular scheduling algorithm is that process synchronization becomes difficult because all execution times are required in advance. The solution devised in SRT was to split a process into sub-processes at the semaphore wait statement; these sub-processes could then be scheduled independently. When a sub-process passes the wait semaphore the next consecutive sub-process is added to the queue for scheduling. The disadvantage of this method is that the scheduler cannot guarantee to meet the constraints of the whole process until the last sub-process has been added to the process queue.

4.3. Advantages and Disadvantages of the Two Algorithms

The rate monotonic algorithm only knows when a process has failed when its time expires, thus wasting valuable processor time on a process which is inevitably going to fail. The bin-packing algorithm looks ahead and determines if the current load can be scheduled; if not the scheduler knows that it will fail long before it actually does so and appropriate action can be taken.

The rate monotonic algorithm requires processes to be given priority according to their frequency rate, the highest frequency having the highest priority. This priority assignment rule is optimum as no other fixed priority assignment rule can schedule a set of processes which cannot be scheduled by the rate-monotonic priority assignment rule (Liu 1973). A process with a low frequency rate will have a low priority regardless of the actual importance of the process.

Real-time system designers favor some variation of the fixed priority scheduling algorithm (Sprunt 1989), for the following reasons:

- it is simple and well understood,
- it has few run-time overheads because the selection of tasks is a simple function,
- it is easier to validate than the dynamic scheduling policies.

Timing requirement should be met by some well understood scheduling algorithm, so that the timing behavior of the system is predictable, understandable and maintainable (Stankovic 1988). Jordan describes the experiences in structuring software in a hard real-time environment using a fixed priority scheduler (Jordan 1990).

The bin-packing scheduler has a larger overhead as it has to calculate if a process can be scheduled so that its timing constraints are met. This algorithm is also generally better than the rate monotonic algorithm at scheduling large number of sporadic processes whereas the rate monotonic algorithm is better at scheduling periodic processes.

The next generation of real-time systems may require adaptive scheduling algorithms which can deal with highly dynamic environments (Stankovic 1988). Fixed scheduling algorithms help provide predictability but have many disadvantages when dealing with an unpredictable environment.

4.4. Determining which Scheduling Policy to Use

Scheduling policies are generally mathematically proven so as to guarantee certain characteristics of the algorithm. Real-time systems can be very complex so selecting an appropriate scheduling policy will often force certain rules onto the system. SRT enables a logical configuration to be constructed without being influenced by a particular scheduling policy. SRT enables a real-time system prototype to be constructed and then various scheduling policies to be evaluated by simulation to determine the system's behavior. It is easy for the software designer to see how well each policy meets its objective. The software designer need not know how each scheduler policy works but it does enable him to pick the most appropriate one for his system. Once a scheduling policy has been decided, the model may only need fine tuning for performance purposes.

4.5. Scheduler Timing Attributes

There are two timing attributes associated with the scheduler, namely the processor speed and the scheduler overhead. The first attribute is to set the speed of the processor, that is, how many units or clock cycles can run in a second. This value sets a relationship between the actual time values and the unit time. This following formula is used by SRT to convert actual time into unit time.

$$\text{unit time} = \text{actual time } (\mu\text{s})^{*} \text{ processor speed (MHz)}$$

For example, the following calculates the unit time from an actuator which has a delay time of 1 000 μs and a processor running at 0.1 MHz.

$$1\,000\ \mu\text{s}^{*}\ 0.1\ \text{MHz} = 100\ \text{units}$$

The overhead attribute specifies the amount of processor time the scheduler will take up when swapping two processes. The overhead value is in unit time and this code cannot be pre-empted.

Figure 8 illustrates how SRT has been designed to accommodate different scheduling algorithms. The model is composed of sensor, process and actuator entities. The scheduler entity is a special type of process which can be interchanged with various other scheduling entities. The scheduler holder contains a pointer to each algorithm currently available in SRT. The model interacts with the holder, which in turn interacts with the active scheduler entity. The holder also acts as a filter, letting through only the appropriate messages for the active scheduler. The scheduling algorithm to be used is chosen from a system menu.

This model can be extended, so that scheduling policies can be plugged into sub-parts of the process model and then an overall scheduling policy selected for all the sub-parts together. For example, processes with hard and soft deadlines could be executed under different scheduling policies and an overall scheduler would distribute processor time between the scheduling policies.

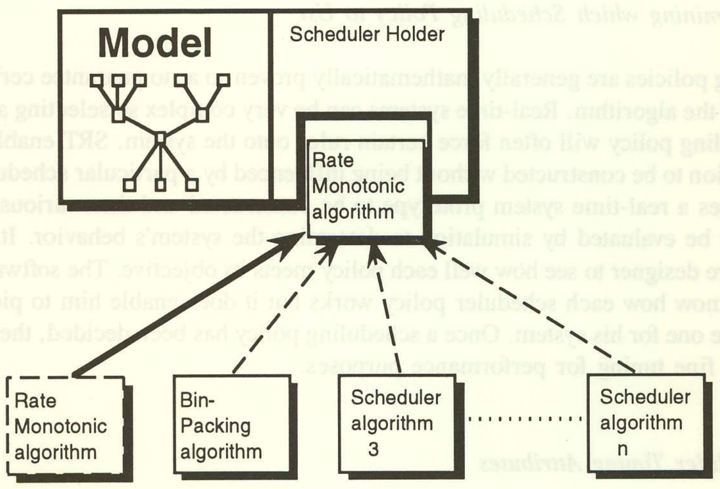


Figure 8. Pluggable scheduler model.

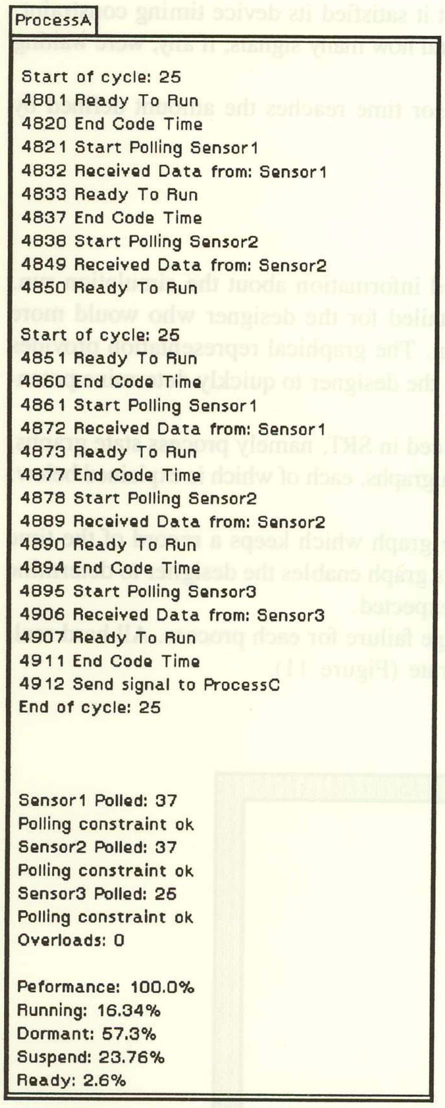
5. Evaluating the Model by Simulation

Once a model of the real-time system has been constructed then the timing constraints can be tested by simulation. The simulation is executed for the unit time specified by the designer. If the simulation is executed in the trace mode then every process reports its progress at every stage. This section looks at the various outputs from the simulation and how they can be interpreted to determine the potential problems in the system. We illustrate the system by comparing the outputs produced using different scheduling algorithms.

5.1. Text Windows

Every process (including scheduler and background) has a text window which displays its actions and the processor time at which the action was started (if in trace mode). Figure 9 shows a scheduler, background and ProcessA's text windows. The scheduler reports to the text view every time processes are swapped and the background process reports the range of times it was being executed. In Figure 9, the process view shows part of the activities of ProcessA during the simulation. On the start up of the 25th cycle the processor executes for 10 units (4851–4860 inclusive). This is followed by the process suspending while sensorl is being polled. Sensorl causes ProcessA to suspend for 12 units, 2 for data transfer and 10 for the sensor delay. This is then repeated for the other two sensors. The final piece of code runs for 5 units, and then the process sends a semaphore signal to ProcessC. After completion of the cycle the process is dormant until its next occurrence. This process has the top priority so no pre-emption will occur during the execution of code. Any scheduling overhead specified would be added to the process every time it is executed.

A summary of the process activities is displayed in each text view. This information can be used to determine which constraints might be modified to improve performance. The



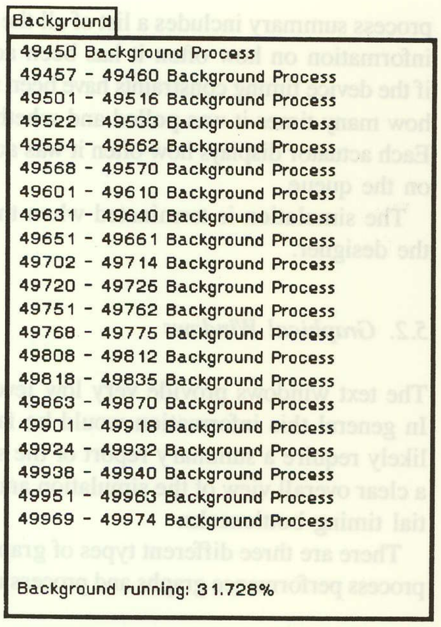
```
ProcessA

Start of cycle: 25
4801 Ready To Run
4820 End Code Time
4821 Start Polling Sensor1
4832 Received Data from: Sensor1
4833 Ready To Run
4837 End Code Time
4838 Start Polling Sensor2
4849 Received Data from: Sensor2
4850 Ready To Run
Start of cycle: 25
4851 Ready To Run
4860 End Code Time
4861 Start Polling Sensor1
4872 Received Data from: Sensor1
4873 Ready To Run
4877 End Code Time
4878 Start Polling Sensor2
4889 Received Data from: Sensor2
4890 Ready To Run
4894 End Code Time
4895 Start Polling Sensor3
4906 Received Data from: Sensor3
4907 Ready To Run
4911 End Code Time
4912 Send signal to ProcessC
End of cycle: 25


Sensor1 Polled: 37
Polling constraint ok
Sensor2 Polled: 37
Polling constraint ok
Sensor3 Polled: 25
Polling constraint ok
Overloads: 0

Peformance: 100.0%
Running: 16.34%
Dormant: 57.3%
Suspend: 23.76%
Ready: 2.6%
```

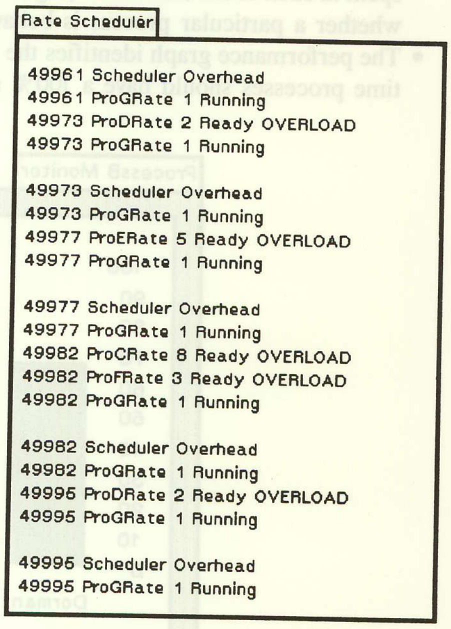
(a) Process View



```
Background
49450 Background Process
49457 - 49460 Background Process
49501 - 49516 Background Process
49522 - 49533 Background Process
49554 - 49562 Background Process
49568 - 49570 Background Process
49601 - 49610 Background Process
49631 - 49640 Background Process
49651 - 49661 Background Process
49702 - 49714 Background Process
49720 - 49726 Background Process
49751 - 49762 Background Process
49768 - 49775 Background Process
49808 - 49812 Background Process
49818 - 49835 Background Process
49863 - 49870 Background Process
49901 - 49918 Background Process
49924 - 49932 Background Process
49938 - 49940 Background Process
49951 - 49963 Background Process
49969 - 49974 Background Process


Background running: 31.728%
```

(b) Background View



```
Rate Scheduler
49961 Scheduler Overhead
49961 ProGRate 1 Running
49973 ProDRate 2 Ready OVERLOAD
49973 ProGRate 1 Running

49973 Scheduler Overhead
49973 ProGRate 1 Running
49977 ProERate 5 Ready OVERLOAD
49977 ProGRate 1 Running

49977 Scheduler Overhead
49977 ProGRate 1 Running
49982 ProCRate 8 Ready OVERLOAD
49982 ProFRate 3 Ready OVERLOAD
49982 ProGRate 1 Running

49982 Scheduler Overhead
49982 ProGRate 1 Running
49995 ProDRate 2 Ready OVERLOAD
49995 ProGRate 1 Running

49995 Scheduler Overhead
49995 ProGRate 1 Running
```

(c) Scheduler View

Figure 9. Text views (a) process view, (b) background view, (c) scheduler view.

process summary includes a list of all the sensor and actuator devices. Each device supplies information on how often it has been activated, so that the figure can be used to check if the device timing constraints have been met. Each sensor connected to the process displays how many times it was polled and whether or not it satisfied its device timing constraint. Each actuator displays how often it was activated and how many signals, if any, were waiting on the queue.

The simulation is terminated when the processor time reaches the amount defined by the designer.

5.2. *Graphical Windows*

The text windows provide very low level detailed information about the simulation run. In general this information would be far too detailed for the designer who would more likely require a summary report of the simulation. The graphical representation provides a clear overall view of the simulation and enables the designer to quickly determine potential timing bottlenecks.

There are three different types of graphs produced in SRT, namely process state graphs, process performance graphs and process utilization graphs, each of which is explained below:

- Each process in the system has a process state graph which keeps a record of the time spent in each of the four states (Figure 10). This graph enables the designer to determine whether a particular process is behaving as expected.
- The performance graph identifies the percentage failure for each process. All hard real-time processes should have a 100% success rate (Figure 11).

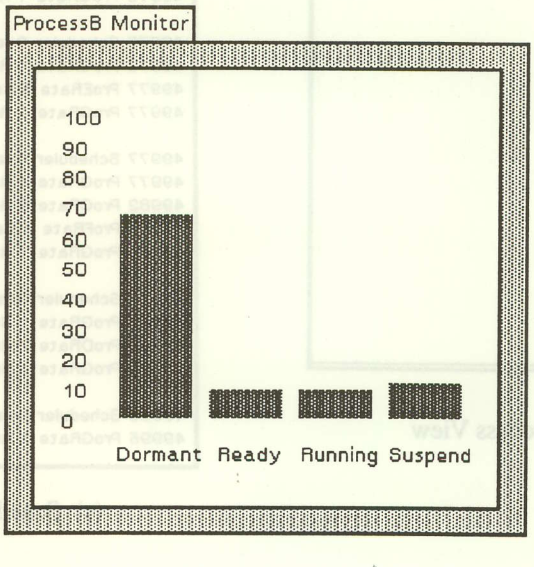
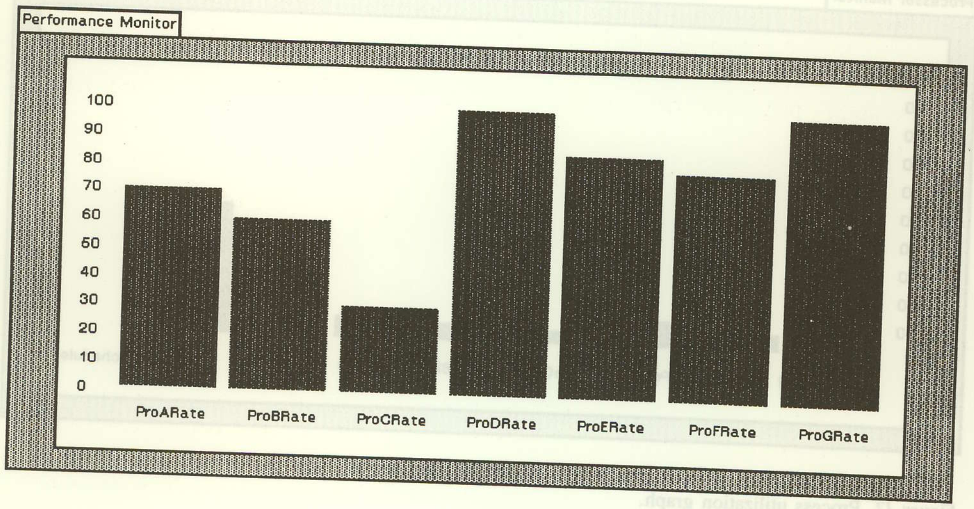
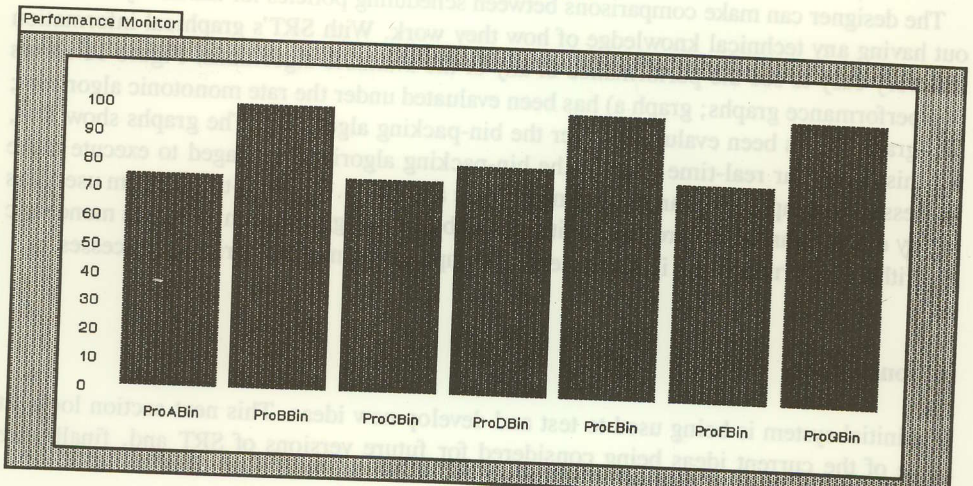


Figure 10. Process state graph.



(a) Rate Monotonic



(b) Bin-Packing

Figure 11. Process utilization graph (a) rate monotonic, (b) bin-packing.

- The process utilization graph identifies how much processor time each process uses (Figure 12). This enables the designer to determine the individual run-times for each process. The percentage of time for which the background process was executed gives the designer an idea of the amount of free processing time remaining. The designer may have a constraint which states that there must be a safety margin of at least a certain percentage of background processing.

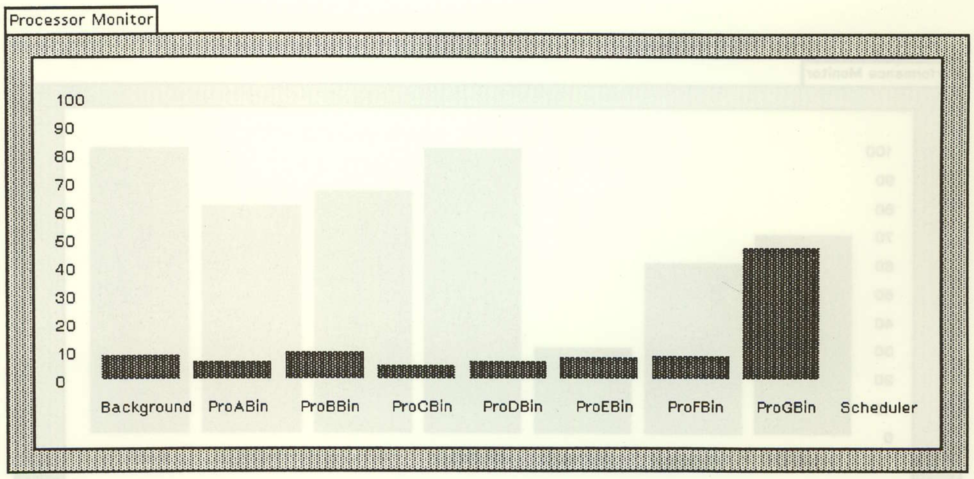


Figure 12. Process utilization graph.

The designer can make comparisons between scheduling policies for his/her system without having any technical knowledge of how they work. With SRT's graphical information it is very easy to see the performance of any of the available algorithms. Figure 11 shows two performance graphs; graph a) has been evaluated under the rate monotonic algorithm; and graph b) has been evaluated under the bin-packing algorithm. The graphs show that, for this particular real-time system, the bin-packing algorithm managed to execute more processes to completion than the rate monotonic algorithm. The real-time system used was highly dynamic and therefore more suited to the bin-packing algorithm. The rate monotonic algorithm performs better if the system is composed of mainly periodic processes.

6. Conclusion

The initial system is being used to test and develop new ideas. This next section looks at some of the current ideas being considered for future versions of SRT and, finally, we draw some conclusions about our work.

The modeling facilities in SRT are continually being improved so that it can be used to tackle real problems. Some of the main ideas are listed below:

- A higher level of abstraction so that a processor could be composed of a number of subsystems and each subsystem could be composed of a number of processes. This would certainly be required if there are hundreds of processes on a single processor.
- Resource modeling. Facilities to make modeling of resources easier. For example, memory could be a separate entity which knows how to interact with other processes.
- Analysis of the simulation data to provide better information about the system so that system bottlenecks can be identified more easily. For example, average and worst-case times of processes could be calculated.

- Extend the process-to-process communication model in order to enable the designer to explicitly incorporate other communication models within SRT, such as the client-server model.

The paper by Blake and Schwan (Blake 1991) which describes the bin-packing scheduling algorithm also explains how the algorithm could be used on a multiple processor system. This could be implemented by adapting SRT to handle a real-time system distributed over many processors. The designer will require the flexibility to be able to move processes to other processors and evaluate the system to endeavor to achieve the best overall performance.

We have demonstrated how SRT can be used to evaluate a model using two very different scheduling algorithms; clearly more scheduling algorithms would be necessary if the tool was to cover the whole of the real-time software application domain. SRT has been developed to accommodate for other scheduling policies but is nontrivial for a designer to add his/her own policies because knowledge of Smalltalk-80 would be required. Future research will look at implementing a tool which would enable designers to define their own algorithms. This tool would require a language (maybe a subset of Smalltalk-80) with which to describe the algorithms. The tool could also be used to produce variations of the scheduling policies already in the system with minimal effort.

The system designer must make estimates on the execution times of processes, which will probably be far from accurate. Project managers, even after years of experience, don't seem to make much better estimates. The estimates are generally made and then multiplied by some factor determined by previous underestimates. SRT does enable changes to be made very easily, so that the consequence of improved estimates can be seen instantly. SRT enables designers to try out various scenarios thus giving them the opportunities to put extreme timing constraints on parts of the system to see how it behaves, for example, if the main data bus has twice the amount of initially estimated traffic. SRT helps designers evaluate scheduling policies so as to determine what policy would perform better for their particular system. Generally the designer will only evaluate a small group of policies which conform to a certain criteria determined by their establishment.

Testing time can be greatly reduced by introducing timing into the system at an earlier stage and having available support tools to determine whether the timing constraints within the design are being met. Nearly 50% of the cost of developing a hard real-time system is spent in the testing phase (Kopetz 1991) so a reduction in this would have significant benefits.

Acknowledgment

The authors wish to thank Paul Kearney, Bill Jones and Morag Kellaway from British Aerospace for their constructive ideas in the development of the current version of SRT.

References

Allworth, S.T. 1981. *Introduction to Real-Time Software Design*. London: MacMillan.

Arvind, K., Ramamritham, K., and Stankovic, J.A. 1991. A local area network architecture for communication in distributed real-time systems. *Journal of Real-Time Systems*. 3: 115–147.

Auernheimer, B., and Kemmerer, R. 1986. RT-ASLAN: A specification language for real-time systems. *IEEE Transactions on Software Engineering*. 12: 879–889.

Berry, G., Moisan, S., and Rigault, J.-P. 1983. ESTEREL: Towards a synchronous and semantically sound high level language for real-time applications. *Proceedings IEEE Real-Time Systems Symposium*. Arlington, VA: IEEE Computer Society Press, pp. 30–37.

Berryman, S.J., and Sommerville, I. 1991. Modelling real-time constraints. *IEE 3rd International Conference on Software Engineering for Real-Time Systems*. IEE, Circencester, UK, pp. 164–169.

Blake, B.A., and Schwan, K. 1991. Experimental evaluation of a real-time scheduler for a multiprocessor system. *IEEE Transactions on Software Engineering*. 17: 34–44.

Dasarathy, B. 1985. Timing constraints of real-time systems: constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*. 11: 80–86.

Gehani, N., and Ramamritham, K. 1991. Real-time concurrent C: A language for programming dynamic real-time systems. *Journal of Real-Time Systems*, 3: 377–405.

Halang, W.A., and Stoyenko, A.D. 1990. Comparative evaluation of high-level real-time programming languages. *Journal of Real-Time Systems*. 2: 365–382.

Harel, D. 1988. On visual formalisms. *Communications of the ACM*. 31: 514–530.

Jahanian, F. 1989. Verifying properties of systems with variable timing constraints. *Proceedings IEEE Real-Time Systems Symposium*. Santa Monica, CA: IEEE Computer Society Press, pp. 319–328.

Jahanian, F., Lee, R.S., and Mok, A.K. 1988. Semantics of modecharts in real-time logic. *Proceedings 21st Hawaii International Conference*. Hawaii.

Jahanian, F., and Stuart, D. 1988. A method for verifying properties of modechart specifications. *Proceedings IEEE Real-Time Systems Symposium*. Huntsville, AL: IEEE Computer Society Press.

Jordan, J.E. 1990. Experiences structuring software in a periodic real-time environment. *Software—Practice and Experience*. 20: 707–718.

Kligerman, E., and Stoyenko, A.D. 1986. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*. 12: 941–949.

Kopetz, H., Zainlinger, R., Fohler, G., Kantz, H., Puschner, P., and Schutz, W. 1991. An engineering approach to hard real-time system design. *European Conference on Software Engineering*. Berlin/New York: Springer-Verlag, pp. 166–188.

Lauber, R.J. 1989. Forecasting real-time behavior during software design using a CASE environment. *Journal of Real-Time Systems*. 1: 61–76.

Lin, K.J., and Natarajan, S. 1988. Expressing and maintaining timing constraints in FLEX. *Proceedings IEEE Real-Time Systems Symposium*. pp. 96–105.

Liu, C.L., and Layland, J.W. 1973. Scheduling algorithms for multiprogramming on a hard-read-time environment. *Journal of the Association for Computing Machinery*. 20: 46–61.

Luqi, 1988a. Knowledge-based support for rapid software prototyping. *IEEE Expert*. 3: 9–18.

Luqi, and Berzins, V. 1988b. Rapidly prototyping real-time systems. *IEEE Software*. 5: 25–36.

Luqi, Berzins, V., and Yeh, R.T. 1988c. A prototyping language for real-time software. *IEEE Transactions on Software Engineering*. 14: 1409–1423.

Luqi, and Ketabchi, M. 1988d. A computer-aided prototyping system. *IEEE Software*. 5: 66–72.

Malcolm, B. 1989. A large embedded system project case study. *Software Engineering for Large Software Systems*. Elsevier Applied Science, Watershed Media Centre Bristol, pp. 96–121.

Orr, R.A., Norris, M.T., Tinker, R., and Rouch, C.D.V. 1988. Systematic method for realtime system design. *Microprocessors and Microsystems*. 12: 391–395.

Sommerville, I. 1992. *Software Engineering*. 4th Edition. Reading, MA: Addison-Wesley.

Sprunt, B., Sha, L., and Lehoczky, J. 1989. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*. 1: 27–60.

Stankovic, J.A. 1988. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*. 21: 10–18.

Stuart, D.A., and Clements, P.C. 1991. Clairvoyance, capricious timing faults, causality, and real-time specifications. *Proceedings IEEE Real-Time Systems Symposium*. San Antonio: IEEE Computer Society Press, pp. 254–263.

Zave, P., and Schell, W. 1986. Salient features of an executable specification language and its environment. *IEEE Transactions on Software Engineering*. 12: 312–325.