A Scheduling Simulator for Real-Time Distributed System*

Antonino Casile, Giorgio Buttazzo, Gerardo Lamastra and Giuseppe Lipari
Scuola Superiore S. Anna
Via Carducci 40
56124 – Pisa, Italy
{casile,giorgio,lamastra,lipari}@sssup.it

21st December 1998

Abstract

This paper describes a tool for the design, simulation and analysis of real-time systems, ranging from simple scenarios with a few types of tasks on a single node to very complex architectures consisting of multiple nodes interconnected with a network and sharing a set of hardware and software resources. The tool includes a discrete event simulation library, a scripting language for describing the system load and the relevant measures, and a graphical interface providing user-friendly interaction with the simulator.

1 Introduction

The design and the implementation of large, distributed real-time systems is a very complex matter. Thanks to all the recent theoretical advances in the real-time research, it is possible to perform a schedulability analysis and verification of the temporal constraints of a real-time application using automatic tools, at least for a uniprocessor environment. Even if it is possible to analyze simple distributed systems using a pure theoretical methodology, when trying to perform a schedulability analysis for real applications, we end up very easily with an NP-hard problem. Moreover, there is no complete theory about statistical information on deadline miss or end-to-end delays for soft real-time applications. So, we believe that a simulation and modeling environment can be very useful, especially during the design phase, to evaluate in a fast and easy way different approaches.

The key ideas behind this project come from the experience acquired by developing a uniprocessor scheduling simulator, GHOST [8], and from the implementation of HARTIK [2, 6], a hard real-rime kernel for critical control applications.

The main objectives that drove this project are:

Portability – It is important for implementing the same application on different platforms, with little effort, saving development time and cost.

Flexibility – The simulator allows the user to develop and experiment novel task models, kernel features, scheduling algorithms, concurrency control protocols, distributed protocols, and so on, in a modular fashion.

*Work carried out under the financial support of the Ministero dell'Universitá e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the Project "Design Methodologies and Tools of High Performance Systems for Distributed Applications"

Easy to use – The tool already incorporates many built-in algorithms and models to allow non-expert users to run their experiments with minimum effort.

1.1 Related Works

Several other simulation tools have been developed in the literature, but most of them do not meet all the requirements stated above. The major lack of these tools is flexibility. There are very few tools which allow end-users to define their own scheduling policy, task model and so on in an easy way.

For example, Kuo and Mok [4] and Song [9] tested some specific protocols through simulation but their work can hardly be extended to include other approaches.

DRTSS is a general tool [11] which has a single formalism to describe both the scheduling policy and the task set, thus allowing a user to specify his own. Unfortunately DRTSS is not oriented to the performance analysis of scheduling algorithms but to modeling and verification of real-time applications.

A more general tool for the simulation of hard real-time systems is STRESS [1], which provides a description language for modeling the system itself and for creating task sets with predefined features. However, it focuses exclusively on tasks with hard timing requirements, without explicit support for hybrid task sets.

Tokuda et al. [14, 15], developed a simulator too, but the tool is specific for the ARTS and the Mach environment and it cannot easily be extended to work on different systems.

A work by Lai and Callison [5] is the most closely related to the present one. This tool allows the simulation and analysis of a set of concurrent tasks which access shared resources with a user-defined concurrency control protocol. Its interface is object-oriented and the user can add his own classes in order to meet his needs. However, no user customization is allowed for the task model.

The Ptolemy tool from the University of Berkeley [3] is an excellent general purpose environment for simulation and synthesis of complex hardware/software systems; it offers different models of computation to cover various application domains. However, its discrete-event simulation domain has been designed essentially to simulate digital hardware. So it is quite difficult to model operating system level algorithms without rewriting or extending the application domain.

The MODSIM III tool from CACI product company is another simulation environment which offers many interesting features: object oriented language, a graphical interface, and an extended library of simulation objects. However, being a commercial product, this tool is not open and requires to learn its specification language if you want for modeling something which is not already present in the library.

2 Simulator Structure

Analyzing the different approaches presented in literature, it emerges that they are either dedicated to the solution of a specific problem or too generic and difficult to use for the simulation of distributed real-time software.

To meet flexibility and expandability, our simulator was designed according to an object-oriented methodology. The simulator core is written in C++, whereas the graphic interface is written in Java. Both modeules are not tied to a specific hardware architecture or operating system, thus allowing full portability.

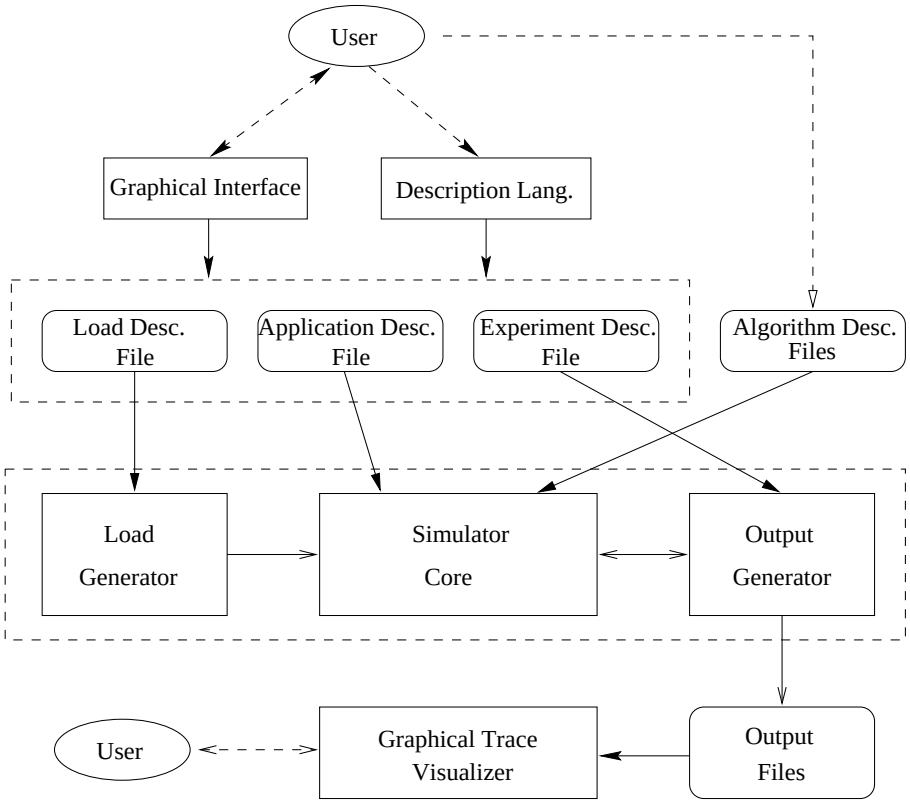

Figure 1: Logical structure of the simulator.

Our aim is to offer different levels of interface in order to allow the user to specify a system at increasing level of complexity. For example, if a user needs to simulate the behavior of a distributed system based upon standard objects, he/she has to specify only the number of nodes, select the scheduling strategies in each node, select the communication protocol and specify the application tasks. On the other hand, if a user wants to simulate the performance of a novel scheduling strategy, he/she has to write a C++ file with the appropriate code: this is easy to do because the simulator provides a library of classes which implements many different objects, like schedulers, task models, resource access policies and communication protocols. Figure 1 illustrates the logical structure of the simulator.

There are three different ways for specifying a system in our simulator. In the first way, the system structure can be described through a *Graphical Interface*, written in Java, which allows the user to select the more appropriate objects, build their model, and set their properties. Many system parameters can be selected from another window to monitor the simulation. The user can change parameters, simulate the system and use the trace visualizer without leaving the graphical environment, so there is an immediate feedback on what is happening, that is very helpful.

The second way for specifying a system is through a description file. The description language is more explicit and is thought for machines which do not support the JAVA runtime environment. The graphical interface and the description language have the same expressive power, and the the description can be automatically obtained by the graphical representation.

Before starting a simulation, the description file (or the graphical representation) is processed and translated into a set of C++ files, which are compiled and linked to the simulator

library.

Generally, three files have to be produced before running any simulation:

- an *Application Description File*, which specifies the structure of the system in terms of nodes and network connections;
- a *Load Description File*, which describes the desired characteristics of the application tasks in terms of computation times, deadlines, activation rates, and so on; each of these parameters can also be expressed as a random variable with given statistical distribution;
- an *Experiment Description File*, which contains the parameters that have to be monitored during the simulation.

Alternatively, it is possible to directly write a C++ code, link it with the simulator core libraries, and run the simulator just as any other executables.

The third way for specifying a system is at the C++ level. At this level the user can define new scheduling algorithms and new resources and network protocols writing one or more C++ files (*Algorithm Description Files*). Clearly, this is more difficult to do, since the knowledge of the internal structure of the simulator is required. However, the object-oriented approach simplifies the job, as it is possible to derive the new classes from the existing class library provided by the simulator. Moreover, the encapsulation guarantees that users do not need to know all the internal details, but only the external interface of the classes. This approach facilitates expandability and customization of the tool to the user needs.

3 The C++ Class Library

The library is composed by two set of classes;

- The classes which implement the discrete event simulation engine and the data collection facilities.
- The classes used to describe the system model and architecture;

The core of the tool is the *Simulation* class which processes the global event queue. Any event is an object deriving from the *BaseEvt* class (see Figure 2); it has an associated expiration time, an action which occurs when the event is processed, and a priority. This is necessary, since it is very important that simultaneous events are processed with the right priority order; for example, think of an event of task termination and deadline miss; if the deadline miss was processed before task termination, then the simulator would signal the overflow even if the task is completing exactly on the deadline boundary. Each event has also a source and a target associated with it, representing the object which has posted the event request and the object which will handle it.

The tool can perform different types of simulation; basically it is possible to do a deterministic simulation, with a well-defined behavior for any object in the system, or a stochastic simulation, using the method of independent replications. A robust random number generator is included for this purpose in the source code. The *run()* method of the *Simulation* class basically executes a loop, extracting the first event from the queue and executing its action method. The loop lasts until the specified simulation time is reached. If required, it

```
class BaseEvt : public DListNode<BaseEvt> {
  protected:
    Tick time;
    Entity *target;
    Entity *source;
    EvtId priority;
    static Trace *trc;
    DRefStack<BaseStat> stats;
    static DQueue<BaseEvt> evtQueue;

  public:
    Bool trace;

    BaseEvt(Entity *f, EvtId evtType);
    void post(Entity *dest, Tick myTime);
    void drop(void);
    BaseEvt *getFirst(void);

    EvtId getPriority(void);
    Tick getTime(void);
    Entity *getSource(void);
    Entity *getTarget(void);
    void addStat(BaseStat &actStat);
    virtual void action(void);
};
```

Figure 2: BaseEvt class definition.

```
class StatMean : public BaseStat {
  protected:
    double count;
  public:
    StatMean(double i) : BaseStat() {
      val = i; count = 0;};
    virtual void record(double a) {
      val = val * count + a;
      val / = ++count;
    };
};
```

Figure 3: Example of probe.

more replications can be executed to perform a statistic analysis and compute the confidence interval for the variables under observation.

The tool allows, for example, to separate the design and the analysis phase; it is possible to design a new scheduling algorithm or a new task model and test its performances easily without inserting probes in the code or using an already assessed test-bed. This is possible because all the interesting data are collected at the event processing layer. The simulator can produce two kind of output: a trace file, representing the actual execution, and a set of statistics over different system parameters.

The tracing mechanism is implemented with the *BaseTrace* class; this class is just an interface for a prototype tracer module. A standard implementation is also provided, together with a graphical visualizer, written in Java. Whenever an object has to trace its execution, it just links the *Trace* class to its events. Then the event processing code will call the *record()* method of this class to record the event. The trace class can be user-defined to cope with the specific user needs or to allow the use of a different visualizer.

The available graphic visualizer (see Figure 4) displays a representation of the system evolution in a desired time scale. The user has the possibility to move forward and backward along the trace, change the scale factor (zoom), go to the time at which a particular event occurred, and choose the parameters to be represented. The user can also obtain additional information on the trace, such as event types and task properties (for example, deadlines and periods).

Statistic is based on a set of basic probe objects all derived from the *BaseStat* class. This class implements all the functions necessary for doing the analysis; it is initialized with the number of experiments to be done, and has an array where data are recorded at the end of the simulation. It has two pure virtual methods, *probe()* and *record()*, used during the collecting phase, and some methods used to obtain the final value. The probes can be easily defined specifying the method *record()*; for example, it is possible to define a mean probe by deriving a class as shown in the code fragment of Figure 3.

Some basic probes (min, max, mean and others) are provided.

In this way the user needs only to write the code to specify what has to be observed. This can be done deriving a class from the needed probe and specifying the method *probe()*; for example:

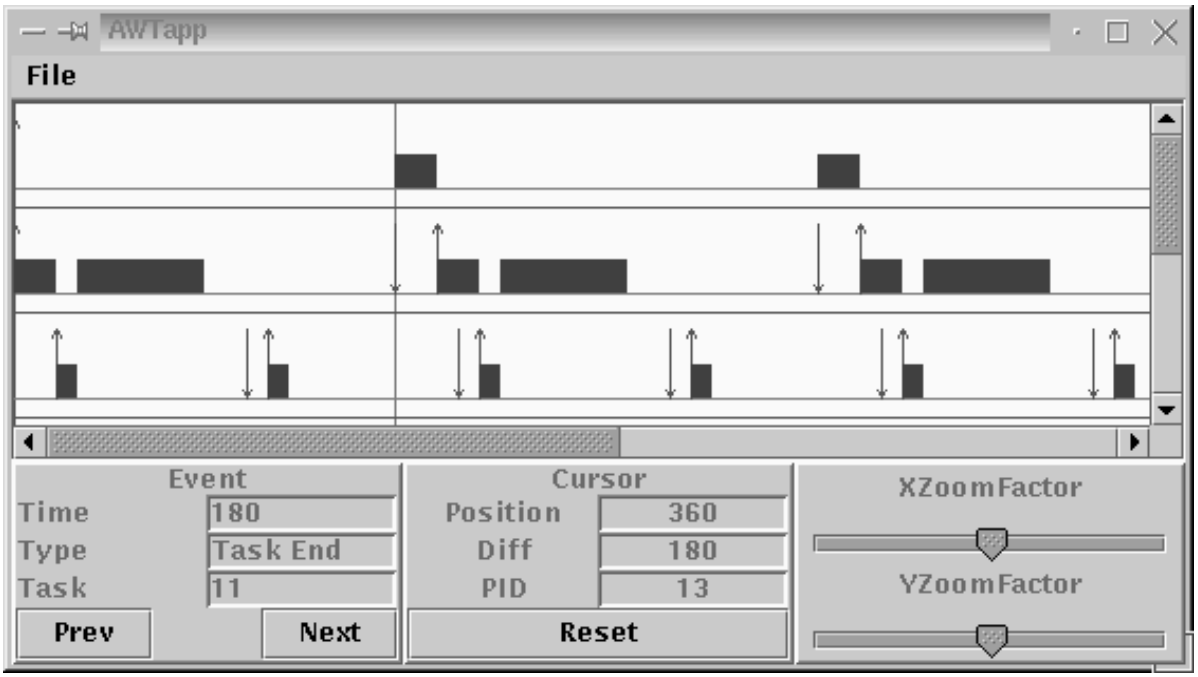

Figure 4: A screenshot from the graphic visualizer.

```
class StatFT_Task1 : public StatMax {
   ...
   virtual void probe() {
      record(task1.release);
   };
   ...
};
```

Each event is associated with a list of statistical probes which are invoked when the event is processed. Whenever an event is processed, every statistical probe attached to it is invoked and data is saved in a table. When a given replica is terminated, any statistical probe in the system is updated to produce the final value. When a complete run (composed by several replicas) is ended, data are stored in a file and the confidence interval is evaluated.

As stated before, it is possible to build a complex system by combining objects representing the system components. Each object which needs to be simulated is derived by the *Entity* class.

An Entity has essentially three characteristics:

- An internal state, which can be changed by direct method invocation or as a consequence of an event handling;
- The set of the events which the entity can generate;
- An event handler which the Entity needs to process.

By using the C++ inheritance, it is possible to build entities having a well-defined behavior and with an arbitrary level of detail. The *Entity* class definition is shown in Figure 5.

```
class Entity : {
  protected:
    Entity *father;
  public:
    virtual void eventHandler(BaseEvt&,
      const Tick time, Entity *from) = 0;
    virtual EntId getEntId(void) = 0;
    virtual void newRun(void) = 0;
    virtual void endRun(void) = 0;
    virtual void endSim(void) {};
};
```

Figure 5: Entity class definition.

Entities may be composed into hierarchy trees to model complex systems. Each entity has a field, *father*, which points to its immediate predecessor in the tree. An entity E_1 is said to be *contained* in another entity E_2 if the *father* of E_2 points to E_1. For example, if a node entity contains a list of task entities, the *father* of each of these tasks is the node itself, as in a standard tree structure. If the task needs to be described at a deeper level of complexity, it may contain a list of pseudoinstructions. It is worth noting that, as each task class is derived from a common *BaseTask* class, it has a standard interface. So, the node entity does not need to know whether the actual task object contains pseudoinstructions or not, as long as they interact using the standard interface. Of course, this is true for entities of any level.

In general it is possible to give an arbitrary level of complexity to the architecture under analysis. Conventionally, four layers have been defined:

- Pseudo-Instructions;
- Tasks;
- Task Servers;
- Nodes.

This choice appeared to be sufficient for modeling most different systems.

Each layer of this system has its own events, but it is possible that an entity of an upper layer needs to handle the events generated by the entities it contains. So, when the event action is executed, the control is usually gathered by the entity which generated the event; it then forwards the event to the upper layer entity; this happens until the father of the entity is null.

On the opposite side, the entity of an upper layer cannot directly invoke the event handler of a contained one, but it must communicate with it using its public available methods.

The modeling of distributed architecture is based on a simple interface between the *Node* and the *Network*. These two entities cannot be characterized using the containment relation outlined previously; in fact, since a node can be generally connected to two different networks, it might be impossible to define its father. To model the interactions among Nodes and Networks it is possible to use the capability of a given Entity to post an event which will be handled by a different Entity. To avoid replications of common data, a new class *Message* has been defined. This class contains the message data (source, destination, length and any other

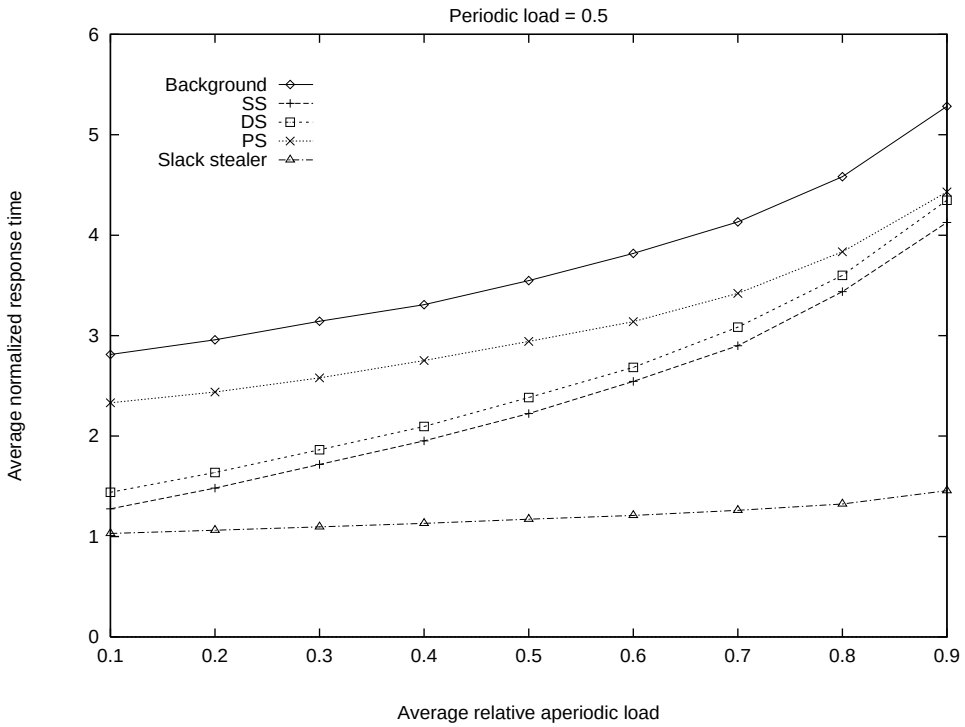

Figure 6: A sample simulation experiment on aperiodic servers under Rate Monotonic.

message parameters) and a set of basic methods defining the message processing. Figure 7 shows the *Message* class definition.

When an entity needs to send a network message, it first creates an object of the Message class, fills its internal data and then calls the method *post()* of the *Message* class. The method posts an appropriate event to the entity which will handle it. The message will traverse all the involved entities and eventually it will be forwarded through the network. It is worth noting that it is necessary to attach an event only to the step which the user wants to process through statistical analysis. So it is possible to define the network transmission protocol at the desired level of complexity.

```
class Message {
  protected:
    Address src, dest;
    int length;
    int virtual post(void);
    int virtual sent(void);
    int virtual delivered(void);
    int virtual processed(void);
};
```

Figure 7: Message class definition.

4 Some Simple Experiments

We have tested the simulator under different design situations. Figure 6 shows an example of output produced by the simulator and plotted using GNU-Plot. It shows the performance results of a simulation experiment on five classical scheduling algorithms for servicing aperiodic requests under the Rate Monotonic priority assignment. In particular, the Background service, the Polling Server (PS), the Sporadic Server (SS) [10], the Deferrable Server (DS) [12], and the Slack Stealer [7, 13] are compared in terms of aperiodic average response time for different aperiodic loads. The periodic tasks are specified with periods ranging from 10 to 30, and processor utilization $U_p = 0.5$. The aperiodic requests have a computation time ranging from 0.5 to 2.5 and are generated to create a load from 0.1 to 0.9 of the bandwidth left by the periodic tasks. The servers have been developed independently, and have been easily integrated into the existing rate-monotonic node entity. It was not necessary to write any other source code to obtain the presented results.

5 Conclusions

In this paper we have presented a tool for modeling, simulating and analyzing a broad range of real-time applications, from classical control systems to soft real-time distributed architectures. The user can interact with the tool at different level of complexity, graphically designing his application from standard building blocks or describing new ones using an object-oriented approach. The whole system is designed to be very portable, allowing the user to work on any platform supporting standard C++ and Java compilers. The graphical user interface is extremely useful for rapid system prototyping and for exploring, at an early design stage, the impact of different solutions for network topology or computational load distribution. As a future work, we plan to extend the class library to include the most recent scheduling algorithms and resource protocols developed in the literature for supporting multimedia applications.

References

[1] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Stress: a simulator for hard real-time systems. *Software Practice and Experience*, 24:543–564, 1994.

[2] G. Buttazzo. Hartik: A real-time kernel for robotic applications. In Raleigh-Durham, editor, *Proc. of the 14th IEEE Real-Time System Symposium*, pages 201–205. 1993.

[3] A. Kalavade, J. L. Pino, and E. A. Lee. Managing complexity in heterogeneous specification, simulation, and synthesis. In *Proc. of the IEEE Int. Conf. on Acoustic, Speech, and Signal Processing*, pages 2833–2836. 1995.

[4] T.-W. Kuo and A. K. Mok. Ssp: A semantics-based protocol for real-time data access. In Raleigh-Durham, editor, *Proc. of the 14th IEEE Real-Time System Symposium*, pages 76–86. 1993.

[5] Chih Lai and Rebecca Callison. A framework for simulation of concurrency control policy in real-time systems. In Brookline, editor, *Proceedings of the IEEE Real-time Technology and Application Symposium*, pages 91–99. 1996.

[6] G. Lamastra, G. Lipari, G. Buttazzo, A. Casile, and F. Conticelli. Hartik 3.0: A portable system for developing real-time applications. In *Proc. of the IEEE Real-Time Computing Systems and Applications*, pages 43–50. 1997.

[7] J. P. Lehoczky and S. Ramos-Tuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In Raleigh-Durham, editor, *Proc. of the 13th IEEE Real-Time System Symposium*, pages 110–123. 1992.

[8] F. Sensini, G. Buttazzo, and P. Ancilotti. Ghost: A tool for simulation and analysis of real-time scheduling algorithms. In Raleigh-Durham, editor, *Proc. of the 2nd IEEE Real-Time Education Workshop*, pages 110–123. 1997.

[9] X. Song. *Data Temporal Consistency in Hard Real-Time Systems*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.

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

[11] Matthew F. Storch and Jane W.-S. Liu. Drtss: A simulation framework for complex real-time systems. In Brookline, editor, *Proceedings of the IEEE Real-time Technology and Application Symposium*, pages 160–189. 1996.

[12] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1), 1995.

[13] T. S. Tia, J. W.-S. Liu, and M. Shankar. Algorithms and optimality of scheduling aperiodic requests in fixed-priority preemptive systems. *The Journal of Real-Time Systems*, 1995.

[14] H. Tokuda and M. Kotera. A real-time tool set for the arts kernel. In Raleigh-Durham, editor, *Proc. of the 9th IEEE Real-Time System Symposium*. 1988.

[15] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Towards a predictable real-time system. In *Proc. of the Mach Usenix Mach Workshop*. 1990.