Simulation and Tracing of Hybrid Task Sets on Distributed Systems

Antonino Casile, Giorgio Buttazzo, Gerardo Lamastra, Giuseppe Lipari
Scuola Superiore S. Anna
Via Carducci, 40 – 56124 Pisa, Italy
Tel. +39+50+883451
Fax. +39+50+883215
{casile, giorgio, lamastra, lipari}@sssup.it

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 is based on a discrete event simulation library, a scripting language describing the system load and the relevant measures, and a graphical interface providing user-friendly interaction with the simulator.

Keywords: Simulation, Distributed system, Design Tool

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 [1], and from the implementation of HARTIK [2, 3], 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.

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 [5] tested some specific protocols through simulation but their work can hardly be extended to include other approaches.

DRTSS is a general tool [6] 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 algorithm but to modeling and verification of real-time applications.

A more general tool for the simulation of hard real-time systems is STRESS [7], 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. [8, 9], worked on simulation tools too, but they are specific for the ARTS and the Mach environment and it is difficult to reuse their environment over different systems.

A work by Lai and Callison [10] 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 [11] 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. Anyway, being a commercial product, it is not open and requires to learn its specification language in order to model 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 has been developed in C++, whereas the graphic interface is written in Java. Moreover, they are not tied to a specific hardware architecture or operating system, thus allowing full portability.

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 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 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, set their properties and model the architecture. 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 a more direct way and is thought for system which cannot support the JAVA runtime environment. The graphical interface and the description language have the same expressive power, and 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.

The third way for specifying a system is through C++ code. At this level, the user can define new scheduling algorithms and new resources and network protocols by writing one or more C++ files (*Algorithm Description Files*). Clearly, this is more difficult to do, since a knowledge of the internal structure of the simulator is required. However, the object-oriented approach simplifies the job, as it is possible to derive new classes from the existing ones, provided in the simulator class library. 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.

Once the user has specified the system, the resulting C++ files are compiled and linked to the library, and the simulation is run. One or more output files (specified with the

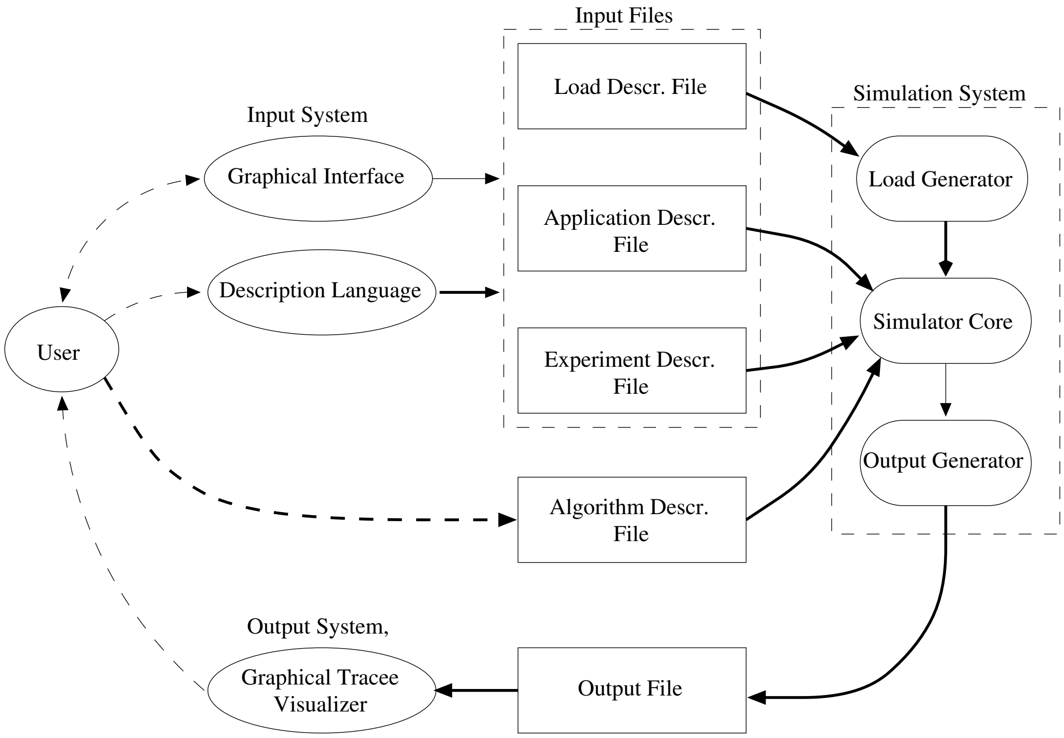

Figure 1. Logical structure of the simulator.

Experiment Description File) are produced. If needed, a *Trace File* can also be produced, which shows the temporal sequence of events in the system during a single simulation run. The trace can be viewed and analyzed with the *Graphical Trace Visualizer*. This is very helpful especially during the test phase of an algorithm or a system.

3. The C++ Class Library

The library is basically composed by two set of classes:

- The first set implements the discrete event simulation engine and the data collection facilities. This set contains the basic classes upon which all the library is based. In the following we will give a closer look at these classes.

- A second set of classes, used to describe the objects which are composed together to form the system model and architecture. This second set contains only classes derived from the first set.

3.1. Events and Entities

Any system is composed by *Entities*, which interact with each other. An *Entity* has an internal state, a set of events which the entity generates and a set of events to which the entity responds.

The simulation is based upon the "discrete events" model: an entity can change its internal state in response to an event. In this way, the state of an entity at any time t_2 can be described by the state at any previous time t_1 and by the sequence of events between t_1 and t_2.

The core of the tool is the class *Simulation* which processes the global event queue. Any event is modeled by an object derived from the *BaseEvt* class (shown in Figure 2). Each event is associated with a triggering time, an action which occurs when the event is processed, and a priority. The **priority** field is necessary, for selecting simultaneous events with the right order. For example, if a task termination event and a deadline miss event arrive simultaneously and the deadline miss is processed before the task termination, then the simulator would signal the miss even although the task completed exactly on the deadline boundary.

Each event has also a source and a target, representing the object which posted the event request and the object which will handle it, respectively.

As stated above, it is possible to build a complex system by combining objects representing the system components. Any object which needs to be simulated is derived by the *Entity* class, showed in Figure 3.

Entities can be composed to create complex objects; this is done through the *containment relationship*. For example,

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

a node *contains* tasks. This relationship is different from standard inheritance. It can be thought as a sort of encapsulation of lower-level entities into a higher-level one. To do so, each entity has a pointer to the entity which *contains* it: this object is referred to as the entity's *father*. Thus, it is possible to create a hierarchy of entities. For example, a task can be described as a simple object or it could be described, at a deeper level of complexity, by explicitly defining the set of pseudo-instructions it would execute during its computation. The pseudo-instructions are contained into the task, so their *father* will be the task itself. It is worth noting that, from the kernel point of view, these two different task objects are handled in the same way, but at a lower level they are completely different. In this way, we take fully advantage of the encapsulation, allowing to easily compose objects without changing their internal implementation. Moreover, it is possible to model a given system at an arbitrary level of complexity. A system is described using four different layers:

- Pseudo-Instructions
- Tasks
- Task Servers
- Nodes

This turned to be sufficient for modeling many different types of real-time systems.

```
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 3. Entity class definition.

When an event is triggered, the control is passed to its *target* entity's event handler. An entity of an upper layer may need to handle the events generated by the entities it contains. So, after handling an event, an entity forwards it to its *father*; this happens until the father of the entity is null. On the contrary, the entity of an upper layer shall not invoke directly the event handler of a contained one, but it must communicate with it using its public methods. The *containment* relationship and the event passing mechanism is shown in Figure 4.

The description of distributed architecture is based upon a simple interface between the *Node* and the *Network* class entities. Their interactions cannot be characterized using the containment relationship previously outlined; in fact, a node may be connected to different networks, then it would be impossible to define a unique father.

The interaction between a *Node* and a *Network* is modeled using their event handling mechanism. In fact, a *Node* can post events whose *target* is a *Network*, and vice-versa. The *Message* class has been defined for this purpose. It contains message data (source, destination, length and any other message parameters) and a set of basic methods defining the message processing. Figure 5 shows the message definition.

To send a message to a network, an entity first creates a *Message* object, fills in its internal data and then invokes its method *post()*. The message may be handled by several entities until eventually reaches the network. In this way, it is possible to model several communication protocols at the desired level of complexity.

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 the independent replications. A robust random number generator is included for this purpose in the source code. The *Simulation* class has a method, *run()*, which executes a loop, extracting the first

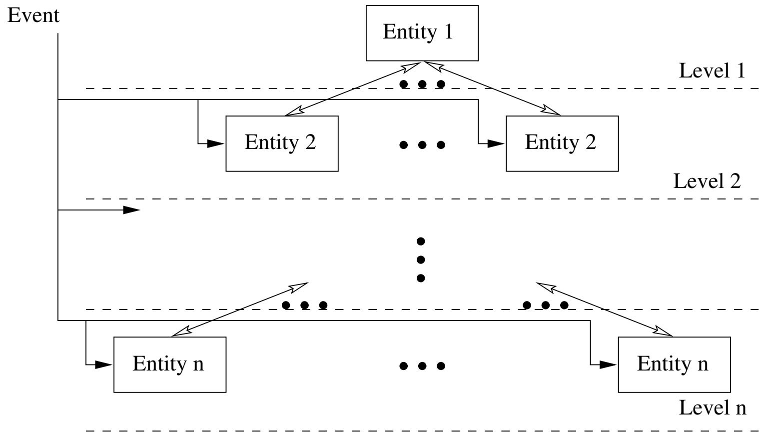

Figure 4. The hierarchical structure of a complex system.

```
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 5. Message 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 6. Example of probe.

event from the queue and executing its *action()* method. The loop ends when the desired simulation length is reached; if it is required, it executes more replications to perform a statistical analysis.

3.2. Trace and Statistics

The simulator can produce two kinds of output: a trace file representing the actual execution and a set of statistics of different system parameters.

The tracing is implemented by the *BaseTrace* class which provides a basic interface to an external graphic visualizer written in Java. In order to trace an event the user only has to set a flag. 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 displays a representation of the system evolution in a desired time scale (see Figure 7). The user can 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).

Statistics are based on a set of basic objects derived from the *BaseStat* class. This class implements a basic interface to do the analysis; it is initialized with the number of experiments to be done, and has an array where the data are collected during the simulation. It has two pure virtual methods, *probe()* and *record()* used during the recording phase, and some methods used to compute the final values.

A set of classes which implement the desired statistics in the *record()* method derives from this class. For example, a class to compute the mean value of a random variable (see Figure 6) is implemented in the simulator library.

The user only has to specify which statistical variables are to be collected. This can be done by deriving new classes from the desired statistical classes and by specifying the *probe()* method. In the following example, a *StatFT_Task* object computes the mean finishing time of a task:

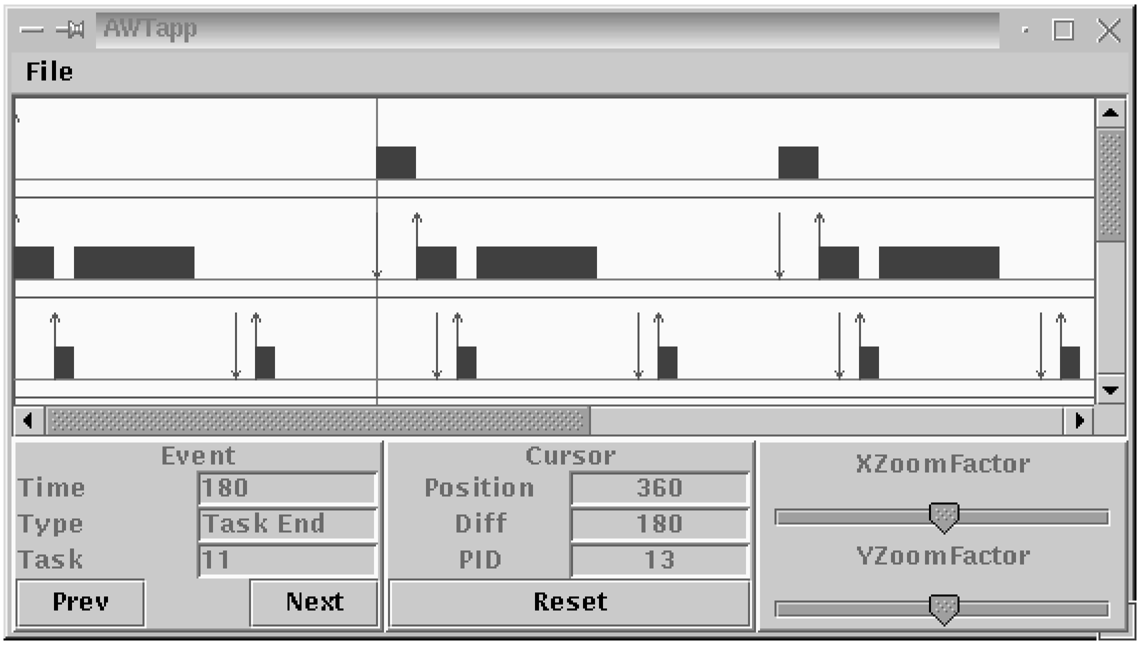

Figure 7. A screenshot from the graphic visualizer.

```
class StatFT_Task : public StatMean {
    Task *task;
    ...
    StatFT_Task(Task *t)task = t;
    virtual void probe() {
        record(task->getFT());
    };
    ...
};
```

Each event is associated with a list of statistical objects. Whenever the event is processed, the *probe()* method of each statistical object attached to it is invoked. All the statistical objects in the system are notified when a replication is over. When a complete simulation run (composed by several replications) is ended, the confidence interval is evaluated and all the collected statistical data can be saved in a file.

This approach allows to nicely separate the statistical evaluation of the state of an object from the actual implementation of the object itself, as long as the object interface is not changed. Hence, for example, if the user implements a new task model, he has to write a new class without worrying about the statistics: he only has to write the methods to get the state of the task. Then, when designing a new simulation, he specifies which variables to measure deriving the suited statistical classes (if needed) and writing the *probe()* method.

4. Experiments

We have tested the simulator under different design situations. Figure 8 shows an example of statistic 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) [12], the Deferrable Server (DS) [13], and the Slack Stealer [14, 15] 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 server have been developed independently, and have been easily integrated into the existing rate-monotonic node entity. It wasn't necessary to write any other source code to obtain the presented results.

5. Acknowledgements

This work has been 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".

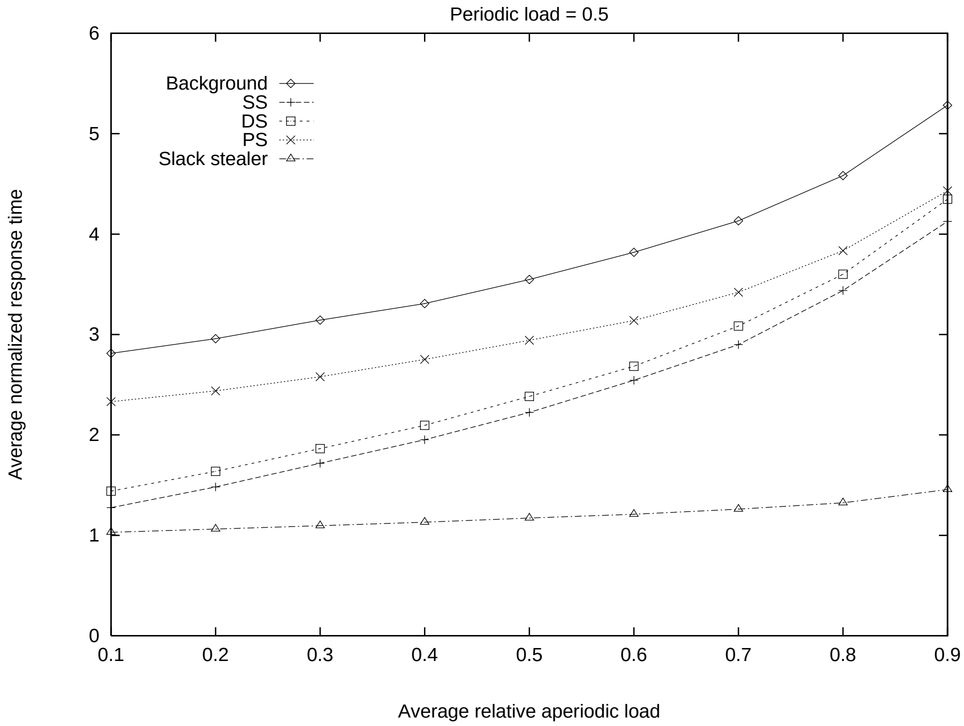

Figure 8. A sample simulation experiment on aperiodic servers under Rate Monotonic.

6. 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] F. Sensini, G. Buttazzo, and P. Ancilotti, "Ghost: A tool for simulation and analysis of real-time scheduling algorithms," in *Proc. of the 2nd IEEE Real-Time Education Workshop*, Raleigh-Durham, Ed., pp. 110–123. 1997.

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

[3] 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*, pp. 43–50. 1997.

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

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

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

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

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

[9] 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.

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

[11] 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*, pp. 2833–2836. 1995.

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

[13] 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*, vol. 44(1), 1995.

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

[15] 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.