Modelling Real-Time Systems

Nancy Lynch
MIT

November 1, 1988

1 Introduction

In order to reason carefully about real-time computing systems, one needs a formal model. Such a model should be usable as a foundation for describing the *problems* to be solved by real-time computing systems, i.e., the requirements specifications. The same model should also be usable for describing (separately from the problems) the *solutions* to these problems, i.e., the algorithms to be used in systems designed to meet the specifications. The model should serve as a mathematical foundation for *correctness proofs*, i.e., proofs that the proposed implementations do indeed satisfy the stated requirements. It should also allow *complexity analysis* of the algorithms. Finally, it should permit *impossibility proofs*, mathematical proofs that demonstrate that certain tasks cannot be performed by any system with particular characteristics. This note addresses the question of what an appropriate model should look like.

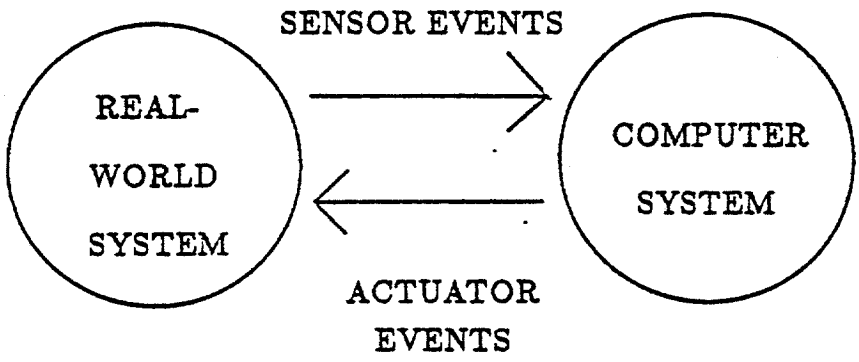
My background is in distributed computing and theoretical computer science; some of my earlier work has involved development of models with the characteristics mentioned above, for distributed systems. I am considering how this work can be extended to apply to this new environment.

2 Characteristics of Real-Time Computing

Real-time systems are distributed systems, but they differ from the systems usually studied in the field of distributed computing in that they generally

consist not just of computer components, but of a combination of computer and real-world components. Distributed systems are usually considered to be interacting with an external environment that provides inputs at somewhat unpredictable times; however, the structure of that environment is generally not considered important. In contrast, in real-time computing, the computer system is considered to be interacting with an environment that is a real-world system, composed of interesting real-world components (such as machines in a factory, or trains and switches). In fact, it is the real-world system that is most important - the purpose of the computer system is simply to provide control signals that cause the real-world system to behave well.

The general architecture of a real-time system looks like:



Interactions between the real-world system and the computer system occur by means of "sensor" events, which convey information about the real-world system to the computer system, and "actuator" events, which convey signals and commands from the computer system to the real-world system. The sensor events generally convey only partial information about the real-world system to the computer system. In addition, the information conveyed may be slightly out-of-date, because of delays within the real-world system. For example, suppose a sensor event is designed to report the arrival of a train

at a nearby station. But by the time a notification of this arrival is transmitted (in the real-world system) to the location of the computer system, the train may have already moved some distance past that station.

Correctness conditions (the problems to be solved) are generally formulated in terms of properties of the real-world system. These conditions are generally of two kinds: *safety conditions*, which say that certain bad real-world system states need to be avoided, and *liveness conditions*, which say that, in certain situations, certain real-world events are required to occur. In contrast to the usual notions of liveness in distributed computing, here one would not normally require just that the desired events should "eventually" occur, but more strongly, that they should occur within a particular amount of time. Sometimes one also requires that they not occur sooner than a specified amount of time later.

An example of an interesting correctness condition is the avoidance of a hazardous state, e.g., a state in which two trains are simultaneously in the same station. Other examples include both "hard" and "soft" real-time requirements - loosely speaking, that certain events either must or should occur by certain times. They also include specifications of the required behavior under conditions of extremely high load, i.e., a specification of the number of real-world events that must be handled in a given time period along with a specification of what to do if this number is exceeded.

Time, of course, plays an important role in real-time computing; it is used in many different ways. A notion of "real time" is fundamental to the description of both the real-world and the computer system. Typically, the activities that occur in the real-world system required certain fairly predictable amounts of time. (I.e., a train travels at speeds that are within certain known bounds, and a machine in a factory processes parts at a certain known rate. Also, the delay between an event π and a corresponding event informing the computer system about π is predictable.) The real-world system may use clocks or timers, whose values or speeds may be within known tolerances of real time; the system may use the arrival of such a clock or timer at a particular value as a signal to take some action. Similarly, the components of the computer system may operate at predictable rates, and the computer system may use clock or timer components to guide its processing.

Failures usually also play an important role in real-time computing. The kinds of failures that need to be considered include timing failures, where an activity that is assumed to require a certain amount of time takes either

too much time or too little. Also, any component, of either the real-world system or the computer system, can fail by stopping, or by omitting certain of its activities, or by behaving in arbitrarily bad ways. Depending on the application, different sorts of requirements may be imposed on the behavior of the system in the face of failures. For many kinds of real-time systems, especially those that involve safety-critical applications, failure-tolerance is crucially important.

A typical organization for a real-time system has the computer system maintaining a fairly explicit simulation of the real-world system. Because of the fact that sensor events only convey partial and out-of-date information, this simulation is necessarily approximate. The simulation can be carried out on a uniprocessor or other tightly-coupled physical system, or on a loosely coupled distributed system. In the former case, the simulation needs to use strategies for processor scheduling with deadlines, while in the latter case, it must contend with the unreliability of communication. (As an example of the latter issue, reconsider the case where a sensor event reports the arrival of a train at a remote station. It would be possible for the computer system to include a computer at the remote station to receive this event; however, then a broken communication link within the computer system could cause serious problems.) A distributed simulation also needs to contend with delays imposed by the particular synchronization primitives used in the language in which the simulation is written. All of these factors - computation delay, communication delay and failure, and synchronization delay - can seriously complicate the job of the computer system, making it difficult for the entire system to satisfy its requirements.

3 Characteristics of a Model

The most important characteristic of an appropriate model is that it be capable of modelling both the real-world system and the computer system, as well as the combination of the two. Ideally, it should do all this in a uniform manner. At the level at which they are normally considered, the computer components are all discrete event systems; for uniformity, then, a model should describe the real-world components in discrete terms as well.

Communication between the computer system and the real-world system occurs by means of sensor and actuator events; sensor events are generated

by the real-world system and transmit information to the computer system, while actuator events are generated by the computer system and transmit information to the real-world system. In most cases, these events are passively received, as interrupts.[1] Again for uniformity, it seems that a similar kind of communication mechanism should be used for describing communication within each of the two subsystems. Thus, a model based on interrupts seems appropriate.

A suitable model needs to include treatment of time and of failures.

A very important characteristic of any model that allows statements about algorithm or system correctness is that it provide a way to express problems and a *separate* way to describe solutions. Although this point may seem obvious, I emphasize it because this separation is not evident to me in some of the published work in this area.

Some examples of models that have been used in the past for modelling real-time systems are timed Petri nets [4], and Statecharts [1].

4 The I/O Automaton Model

My colleagues and I have had very good success using the new I/O automaton model [5, 6] for modelling asynchronous distributed systems; it seems that extensions of this model may be useful for modelling real-time systems as well. In this section, I give a brief summary of the basic definitions of the model, and an example of its use.

Each component of a distributed system is modelled as an I/O automaton. An *I/O automaton* consists of five components: an *action signature*, describing the actions that are *input*, *output* and *internal* to the automaton, a (not necessarily finite) set of *states*, a distinguished subset of the state set that are designated as *start states*, a set of *steps* of the form (state,action,state), comprising the transition relation of the automaton, and a *partition* on the set of output and internal actions, indicating which actions have the same locus of control (e.g., are generated by the same process within the automaton). An I/O automaton is required to be *input-enabled* - an effect is defined for every input action in every state.[2]

[1] In cases where the computer system uses polling, one can model the polling in terms of lower-level interrupts.

[2] In some cases, when inputs arrive at inappropriate times, the automaton might behave

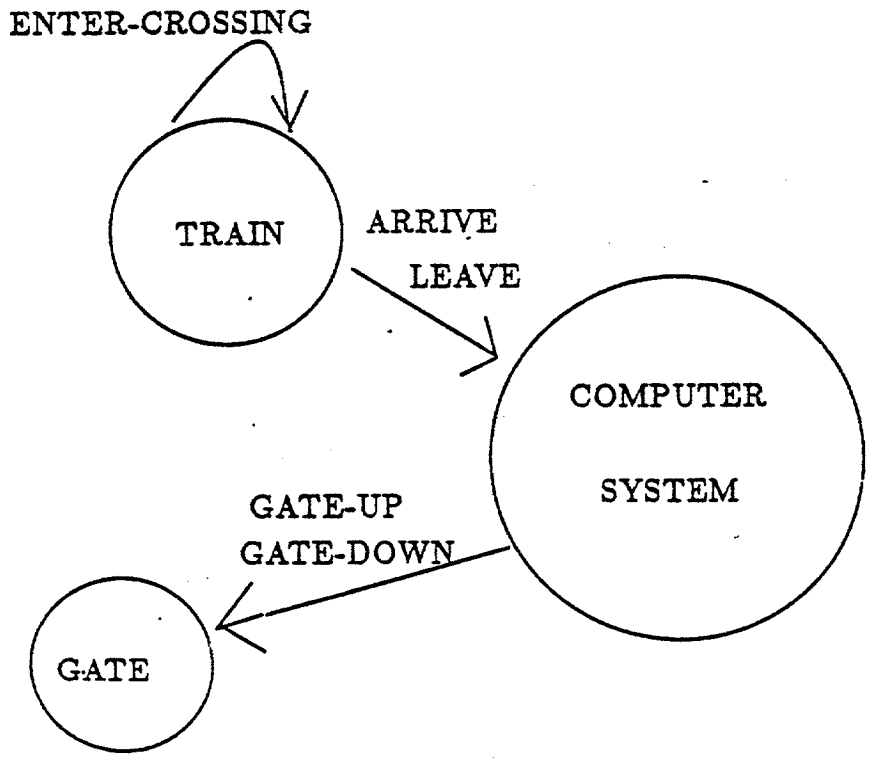
An *execution* of an I/O automaton A is an alternating sequence $s_0,\pi_1,s_1,...$ of states and actions beginning with a start state and with each consecutive (state,action,state) triple a step of A. An execution α is said to be $fair$ if the following hold: (a) if α is finite, then no action is enabled in the final state, and (b) if α is infinite, then either α contains infinitely many events in C, or else α contains infinitely many occurrences of states in which no action of C is enabled. The definition of fairness captures the fundamental idea that each locus of control (class within the partition) gets infinitely many turns to take steps if it "wants to".

I/O automata can be composed to yield other I/O automata. The composition operator is only defined for collections of automata in which there are no shared outputs, i.e. no action is an output of more than one component. However, an action can be an output of one component and an input to any number of other components. The composition automaton synchronizes the occurrences of actions with the same name.

As an example, borrowed from [4], consider a system consisting of three automata, modelling a train, a gate and a computer system respectively.

---

badly, or might detect an error.



The TRAIN automaton has no input actions, has the output actions ARRIVE and LEAVE, and has the internal action ENTER_CROSSING. The states are "not_here", "approaching_crossing", "in_crossing" and "gone", where the initial state is "not_here". The steps are described by the following

ARRIVE
Precondition: state = "not_here"
Effect: state := "approaching_crossing"

ENTER_CROSSING
Precondition: state = "approaching_crossing"
Effect: state := "in_crossing"

LEAVE
Precondition: state = "in_crossing"
Effect: state := "gone"

The partition groups all three actions together. Thus, the train, under its own control, moves through the designated stages, outputting an indication that it has arrived or left.

The GATE automaton has no output or internal actions, but has input actions GATE_UP and GATE_DOWN. The state is simply "up" or "down", initially "up". The steps are as follows.

GATE_UP
Effect: state := "up"

GATE_DOWN
Effect: state := "down"

Here, there are no preconditions because inputs are always enabled.

Finally, the COMP_SYS automaton has input actions ARRIVE and LEAVE, output actions GATE_UP and GATE_DOWN, and no internal actions. The state consists of values for two variables *train* and *gate*, where *train* takes on values in "not_here","here","gone", initially "not_here", and *gate* takes on values in "up","down", initially "~~down~~". The steps are:

ARRIVE
Effect: train := "here"

LEAVE
Effect: train := "gone"

GATE_DOWN
Precondition: train = "here" and gate = "up"
Effect: gate := "down"

GATE_UP
Precondition: train = "gone" and gate = "down"
Effect: gate := "up"

The partition groups the two output actions together.

The composition of the three automata models a real-time system in which the real-world system is modelled by the composition of TRAIN and GATE and the computer system is modelled by COM_SYS. The sensor events are the occurrences of the ARRIVE and LEAVE actions, while the actuator events are the occurrences of the GATE_UP and GATE_DOWN actions. Note that this system has a reachable "hazardous state" in which TRAIN is in state "in_crossing" and GATE is in state "up". (In this state, COM_SYS is in state ("here","up").) In [4], two approaches are discussed for ensuring that the system avoids this state. First, an additional interlock can be added to the system, in this case, having the train wait for a signal from the computer system that announces that the gate is down. Such a strategy can be modelled using I/O automata by modifying TRAIN so that it has a SIGNAL input action, and a second state component "signal" that records whether or not a signal has been seen. The internal action ENTER_CROSSING now should have an additional precondition requiring that signal = "true". Also, COM_SYS would have to be modified to perform a SIGNAL output action when its gate component = "down". The resulting system no longer has any reachable hazardous states. The I/O automaton model is sufficient to describe all the important behavior of this system.

The second approach discussed in [4] is not to modify the system at all, but simply to rely on the timing properties of the real-world and computer subsystems to ensure that the hazard is avoided. Assuming that the time from when a train arrives until it enters the crossing is always longer than the time from when the computer system learns of the train's arrival until it causes the gate to be put down, the bad state will not occur. The I/O

automaton model, as defined so far, does not contain facilities for describing these times, and so cannot be used without modification to describe this approach.

The I/O automaton model has so far been used to describe and verify a substantial number of distributed algorithms, as well as to prove complexity upper and lower bound and impossibility results. Some of the areas to which it has been applied are distributed database concurrency control and recovery, communication, dataflow, distributed consensus, and the implementation of shared registers. The wide range of applicability of this model suggests that it will also be a good starting point for modelling real-time systems.

5 Using the I/O Automaton Model for Real-Time Systems

We expect that an appropriate model for a real-time system will consist of an I/O automaton for the real-world system and another I/O automaton for the computer system. The real-world automaton will have output actions that are sensor actions, used to inform the computer automaton about the state of the real world, and input actions that are actuators originating in the computer automaton. Each of these two automata will in turn be a composition of I/O automata modelling the various components of the two systems.

It is clear that, if the I/O automaton model is to be used in this way, it needs to be augmented to include ways of discussing timing properties. Some preliminary but promising work has already been done in this direction [6]. Instead of considering only executions (alternating sequences of states and actions) of automata, they instead consider *timed executions*, in which real numbers representing times are associated with the occurrences of states in the sequence. (The association of time t with state s means that the automaton entered state s at time t.) The successive times for the states in a sequence must be nondecreasing.

Once time has been introduced into executions, it is possible to define *timing correctness conditions* for such executions. For example, hard and soft real-time requirements can be formulated as properties of timed executions. It is also possible to discuss *timing assumptions* about executions, which

will be needed to guarantee the timing correctness conditions, and may even be needed to guarantee some non-timing correctness conditions (like hazard avoidance in the train example).

Probably the most important category of timing assumptions about executions is that of upper and lower bounds on the time required for certain activities to occur. This category includes, for example, restrictions on the speeds of the various real-world and computer components. It appears that the I/O automaton model can be extended very naturally to support description of this particular category of timing assumptions. What is required is a modification of the definition of *fair execution* so that instead of saying that an action must eventually occur, it instead says that the action must happen within a certain time interval.

In a bit more detail, suppose that each class of an automaton's partition models actions that are under control of the same physical hardware. Normally, that piece of physical hardware has a certain characteristic speed, probably describable as a range of possible times between its actions. (For example, a dedicated computer has a narrow range of times between successive processing steps.) In general, it seems natural to associate with each class in the partition of an automaton, an interval of real numbers describing the allowable amounts of time between successive output or internal actions of that class (or times when no such actions are enabled). Then a *fair timed execution* of the automaton is an execution in which these time bounds hold for all classes in the partition.

With these definitions, I can model the timing-based hazard-avoiding train system above. I simply associate an interval $[l, u]$ with the single partition class in TRAIN, and another interval $[l', u']$ with the class in COM_SYS. This implies that there is a lower bound of l on the time between when the train announces its arrival and when it enters the crossing, and also an upper bound of u' on the time between when the computer learns of the train's arrival and when it causes the gate to be lowered. As long as u' is less than l (not an unreasonable assumption if l depends on the speed of the train and u' on the speed of computer processing), the hazard is avoided.

Another aspect of timing that needs to be modelled is the inclusion of explicit clock and timer components. Such components will be modelled explicitly as I/O automata, e.g., a clock can be modelled as an automaton whose only action is an output TICK. (It may even be convenient to model "real time" in this way, as a special real clock automaton within the system,

but I don't yet see how to make use of this convention.) Assumptions about the timing behavior of these clocks and timers must be described. Some of these assumptions will involve the relative rates of these clocks and timers (and their rates relative to real time); these assumptions could be modelled using the interval ideas described above. There may be other kinds of assumptions about clock components that do not fit so nicely in the framework I have already described, e.g., that the actual values of the clock components remain close to real time.

It is also necessary to augment the model to include modelling of the various notions of failures. This does not seem to be a major difficulty, once we know exactly what kinds of failures are important. For example, failures in which a single component just stops without warning are describable in terms of a modification of the original automaton that has a special *halt* state.

6 Research Directions

6.1 Augmenting the Model

The first work to be done here is to augment the basic I/O automaton model to include time and failures. This work will build heavily on the work already done in [6]. I expect that these modifications will be done in a general form, not specifically tailored to real-time computing, but suitable for many other uses (such as analyzing the time complexity of asynchronous algorithms or modelling timing-dependent communication protocols) as well.

The augmented model needs to be evaluated for its usefulness in modelling interesting real-time systems. For example, we need to discover how generally useful the interval definitions described above are for describing realistic timing assumptions.

Building on this general-purpose augmented model, we should describe whatever we can that is characteristic of real-time systems, e.g., the division into real-world and computer system components. Perhaps we can give a general way of describing the simulations carried out by the computer systems of the corresponding real-world systems, (including the approximate and out-of-date nature of such simulations).

Perhaps there is a general way to characterize the kinds of correctness

conditions normally required for such systems. For example, [2] describes a language, Real Time Logic (RTL), for specifying a wide variety of periodic and sporadic timing constraints; a claim that seems implicit in their work is that this language is sufficiently general to express most interesting correctness conditions for real-time systems. Assuming this is so, it would be interesting to know how to translate RTL specifications into specifications within the extended I/O automaton model.

Some problems to be solved by real-time systems seem to have a static nature (e.g., involving machines with fixed locations processing parts that arrive on conveyor belts), while others seem to be more dynamic (e.g., involving a system of mobile robots). It is interesting to consider whether the same models are appropriate for describing both of these kinds of problems. The I/O automaton model, for example, seems most suitable for statically configured systems; how well does it work for modelling more dynamic systems? Does anything new need to be added to the model to enable it to be used for dynamic systems?

It seems useful to compare the effectiveness of I/O automata to that of other candidate models for real-time systems. In particular, the I/O automaton model should be compared with the timed Petri net model and the Statechart model. Hewitt's actor model has many similarities with I/O automata, but it is more dynamic; perhaps it will provide some helpful ideas for modelling dynamic real-time systems.

6.2 Prototype Examples

Before a useful theory can be developed, I think a better understanding is needed of the basic issues that the theory needs to address. I see a need for the identification of a collection of simple problems that capture the essence of real-time computing problems. Such prototypes can be used as a testbed for different approaches to problem specification and algorithm design, description, verification and analysis. They may even be useful as a source of ideas for basic combinatorial results - upper and lower bound and impossibility theorems describing what can and cannot be done by real-time systems.

I would expect problems to involve simple factory systems, train systems, elevators, airplanes and the like. Some interesting examples in the literature include the train example described above, the Martian Lander in [2] and the

nuclear reactor in [3]. It would be nice to abstract from these applications to describe scenarios that arise in many different applications. Hazard avoidance (while ensuring that the system makes progress) is one example of a general problem. Other problems might involve a computer system issuing the proper signals for different real-world components so that they can all cooperate in accomplishing some coordinated external activity.

For example, the nuclear reactor example of [3], can essentially be abstracted to a simple problem of mutual exclusion, similar to that studied in distributed computing theory. The new twist here, however, is that the real-world system gives no explicit indication to the computer system of when a critical task has been completed; rather, the system used the passage of time to infer that the critical task is done.

The statements provided for these prototype problems should say as little as possible about the organization of the solution. In some cases, where prototype problems are to be derived from examples in the research literature, it will be necessary to restate the problems to say less about the implementation than is done in the original paper.

It would be useful to see if the I/O automaton model is adequate for modelling these prototype problems. (That should provide a good start toward seeing if the model is adequate for modelling more realistic problems.)

As a way of obtaining good sources for such examples, I would like to see a comprehensive bibliography of relevant papers on real-time computing.

6.3 Combinatorial Results

Once prototype problems have been defined, it would be very interesting to prove combinatorial upper and lower bound and impossiblity results about what can and cannot be done by real-time systems, and what costs must be incurred. There are many, many such results in the theory of distibuted computing, and I would expect that similar results should be obtainable for real-time computing.

For example, there is an extensive theory within distributed computing about the complexity, possibility and impossibility of solving mutual exclusion problems. Hazard-avoidance problems seem to be quite closely related to mutual exclusion problems, and so there should be similar results, (or at least similar questions), about hazard-avoidance. Of course, the new setting is quite different from the old because the new setting includes assumptions

and requirements involving time. It should be very interesting to see to what extend the use of time changes the combinatorial results.

Another area of distributed computing theory that has analogies for real-time computing is that of distributed fault-tolerant consensus. (In fact, one of the first distributed consensus problems studied, that of Byzantine agreement, originally arose from real-time computing, in the SIFT project [7].) Synchronization problems seem to be quite closely related to consensus problems, and so again there should be similar results and questions.

Other combinatorial results may arise from the "folk wisdom" already accepted in the real-time computing field. For example, certain real-time systems operate in a very conservative, sequential manner; such a system may be very fault-tolerant but also slow. There may be an inherent tradeoff between fault-tolerance and timeliness, which may be provable as a lower bound result.

6.4 Methodological Results

Once it is known how to describe interesting problems and their proposed solutions within a model, an important effort is to develop general techniques for verifying the correctness of such protocols. Much work has already been done on such techniques; it would be interesting to see if the most important known techniques can be applied to an I/O automaton-based model. Of course, it would also be interesting to develop new techniques, as suggested by the examples.

Ideally, the combination of work in modelling, abstracting and studying prototype examples, proving combinatorial results and carrying out verification will eventually lead to new insights for the design of real-time systems. It's too early for me to speculate very much on what these new insights might be, but I can at least express some initial prejudices here.

I suspect that the computer systems used in real-time systems will become more distributed in the future. In fact, a typical computer system structure might consist of a separate processor for each real-world component, interacting directly with that component; this means that the structure of the computer systems will look a lot like the structure of the real-world system, and the simulation will be fairly direct (though approximate). As the systems become more distributed (and as more parallel hardware becomes available), scheduling issues, previously a focus of much of the work in real-time com-

puting, will no longer be of primary importance. Rather, the focus will shift to communication issues, especially those of efficiency and reliability. The strong kinds of synchronization used in Ada may prove to be an impediment to timely processing; rather, non-blocking message-sending synchronization primitives may prove more usable. Fault-tolerant communication protocols will become increasingly important.

References

[1] Harel, Statecharts: A Visual Formulism, 85,*Science of Programming*

[2] F.Jahanian and K. Mok, Safety Analysis of Timing Properties in Real-Time Systems,*IEEE Transactions of Timing Properties in a Real-Time Systems*, Vol. SE-12. NO. 9, September 1986

[3] F.Jahanian and K. Mok, A Graph-Theoretic Approach for Timing Analysis and its Implementation, *IEEE Transactions on Computers*, Vol. C-36, NO.8, August 1987

[4] N. Leveson and J.Stolzy, Safety Analysis Using Petri Nets, *IEEE Transactions on Software Engineering*, Vol. SE-13 NO.3, March 1987

[5] N.Lynch. I/O Automata: A model for discrete event systems. Technical Memo Massachusetts Institute Technology/LCS/TM-351, Massachusetts Institute Technology, Laboratory for Computer Science, March 1988. Also, in *22nd Annual Conference on Information Science and System*, Princeton University, Princeton, N.J., March 1988

[6] N. Lynch and M.R. Tuttle, Hierarchical correctness proofs for distributed algorithms. Master's thesis, Massachusetts Institute Technology, April 1987. Also, in *Proceedings of 6th ACM Symposium on Principles of Distributed Computation*, pgs 137-151, August 1987. Expanded version available as Technical Report Massachusetts Institute Technology/LCS/TR- 387, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, Ma., April 1987.

[7] J.H. Wensley, SIFT: design and analysis of a fault-tolerant computer for aircraft control.*Proc. IEEE 66*, 10(Oct.1987), 1240-1255

[8] M.Tuttle, M.Merritt and F.Modugno, Time Constrained Automata. In Progress.

Acknowledgements: Thanks go to Nancy Leveson, for taking the time to discuss her views on real-time computing.