Timing Analysis of Distributed End-to-End Task Graphs with Model-Checking

Zonghua Gu

Department of Computer Science
Hong Kong University of Science and Technology

Abstract. Real-time embedded systems must satisfy system-level timing constraints between external sensor inputs and actuator outputs. Real-time scheduling theory can be used to verify that the system is schedulable, that is, no deadlines are missed, but that alone is not enough. Given that the system is schedulable, how to verify that it satisfies system-level end-to-end timing constraints, such as freshness, correlation and separation? To address this question, we adopt the approach of formal modeling and model-checking. Specifically, we use Timed Automata and the model-checker UPPAAL for verification purposes. We have developed generic modeling templates for a class of distributed task systems that can be used as input the the model-checker in order to verify system-level end-to-end timing constraints. We use an application example of distributed real-time control system to illustrate the utility of our approach.

1 Introduction

Real-time embedded systems must satisfy system-level timing constraints between external sensor inputs and actuator outputs. Gerber *et al* [1] identified three classes of *system-level end-to-end timing constraints*:

- *Freshness constraints* bound the maximum time it takes for the data to flow through the system. For example, the time interval between reading input data d from the sensor and writing output data calculated using d to the actuator can not exceed 10ms.
- *Correlation constraints* bound the maximum time-skew between several inputs used to produce an output. For example, if input data d_1 and d_2 are used to produce output data d_3, then d_1 and d_2 must be sampled within 2ms of each other. Generally it is desirable to minimize the data correlation from distributed sensors in order for the controller to have a realistic and consistent view of the physical environment.
- *Separation constraints* bound the maximum jitter between consecutive values on a single output channel.

Given a set of such timing constraints, as well as timing attributes of the tasks composing the system, we would like to verify that the system satisfies

these constraints. Real-time scheduling theory can be used to prove that the system is schedulable, that is, no deadlines are missed, but that alone is not enough. A schedulable system may still violate system-level freshness, correlation and separation constraints. In this paper, we address this question: *Given that the system is schedulable, how to verify that it satisfies system-level end-to-end timing constraints, such as freshness, correlation and separation?*

The growing complexity of modern real-time embedded systems makes it imperative to apply formal analysis techniques at early stages of system development. In this paper, we adopt the approach of formal modeling and model-checking, which is a promising technique for analysis of finite state systems by exhaustive exploration of the system state space. Specifically, we use *Timed Automata* (TA) and the model-checker UPPAAL [2] for this purpose. Our main contribution in this paper is developing modeling techniques for a class of distributed task systems that are used as input the the model-checker UPPAAL to verify system-level end-to-end timing constraints. We make the assumption that the system is already schedulable before applying model-checking. Without this assumption, we would have a much larger state space due to deadline misses, an error condition that should never arise in a production system, without gaining any additional insight into the system's normal operation.

Some authors have developed modeling formalisms and analysis techniques for real-time scheduling analysis. For example, ACSR [3] is a resource-aware real-time process algebra for specification and formal verification of distributed real-time systems. Its main features include the ability to specify resources and their usage by system components, and prioritized execution that allows to express different preemptive and non-preemptive scheduling policies. Similarly, TIMES [4] is based on Timed Automata, and can be used for modelling, schedulability analysis and synthesis of optimal schedules for a set of tasks that are triggered periodically by timers, or sporadically by external interrupts. These approaches have the benefit of being able to deal with more general task models than real-time scheduling theory, which is typically restricted to periodic task models, and has to make pessimistic assumptions when dealing with sporadic or aperiodic tasks. However, modeling real-time scheduling inevitably introduces a much larger state space caused by interleaving of task executions on a shared processor, and has a negative impact on the scalability of model-checking, which is already the most important limiting factor for its industry adoption. In contrast, we adopt an approach of separation of concerns. We do not consider real-time scheduling issues caused by interference among tasks sharing a processor. We either assume a distributed system, where each task has its own dedicated processor, or a system with shared processors, but the timing attributes of tasks already take into account interference caused by multiple tasks sharing one processor, i.e., we use *worst-case response time* instead of *worst-case execution time* as a task's timing attribute. By using real-time scheduling theory to verify the schedulability assumption, and model-checking to verify system-level timing properties, we can achieve much better scalability than using model-checking to check for both schedulability and system-level timing properties.

This paper is structured as follows. In Section 2, we present our system modeling techniques. In Section 3, we use an application example to illustrate the complexity involved in timing analysis of distributed end-to-end task systems, and apply our modeling and model-checking approach to its timing analysis. In Section 4, we discuss related work, and in Section 5, we draw some conclusions.

2 System Modeling with TA

A *Timed Automaton* (TA) is a standard finite-state automaton extended with a finite collection of real-valued clocks, which proceed at the same rate and measure the amount of time that has elapsed since they were last reset. The model-checker UPPAAL [2] adds a few extensions to the standard definition of TA, such as integer variables, CCS-style [5] synchronous communication, urgent channels, etc. For space limitations, we do not provide a detailed description of TA and UPPAAL.

We define the system under analysis with an Asynchronous Task Graph [1], which is a *Directed Acyclic Graph* (DAG), where vertices denote tasks, which have input and output data ports. Directed edges denote dataflow communication between tasks. Tasks execute asynchronously and communicate with each other through shared buffers of size 1. Reading and writing are both non-blocking, i.e., writing a data token to a shared buffer overwrites the previous data token contained in it. Reading and writing are both assumed to be instantaneous, hence we do not require the use of mutual exclusion mechanisms such as mutex or semaphore. Forks and joins in the task graph are of type *AND*. That is, when a task executes, it reads in one data token from each of its input ports in an instantaneous, atomic action at the start of its execution, and then writes out one data token to each of its output ports in another instantaneous, atomic action at the end of its execution.

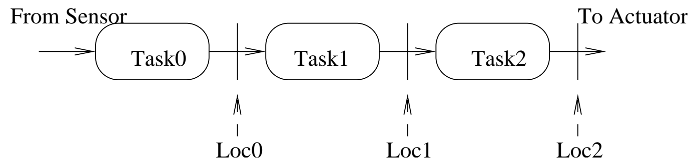


Fig. 1. An end-to-end task graph consisting of a linear chain of sub-tasks, with *atomic, non-interleaving* task execution.

Consider a linear chain of sub-tasks in an end-to-end task graph from external stimulus from the environment to eventual output to the environment, as in Figure 1. The taskset is $\{T_0, T_1, \ldots, T_{N-1}\}$, where T_{i-1} is the immediate predecessor of T_i, where $i = 1, \ldots, N-1$. Since tasks communicate through shared buffers, the system events that we are concerned with are the read/write events

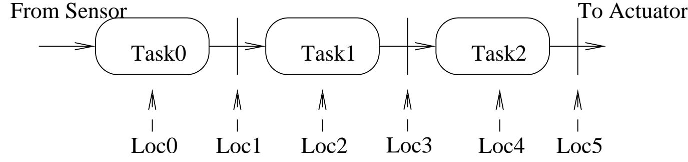


Fig. 2. An end-to-end task graph consisting of a linear chain of sub-tasks, with *non-atomic, interleaving* task execution.

of the shared buffers. Suppose that we know the execution time intervals of each task block, and we want to model this task chain with timed automata. This is straightforward if the execution is single-rate, and there is no pipelining, i.e., each invocation of the task chain executes from start to finish before the next invocation. However, the situation becomes more complicated if *pipelined* and *multi-rate executions* are allowed. The execution is pipelined if it is possible for new inputs to come in from the environment before the previous output has been produced. The execution is multi-rate if it is possible for each task's execution rate to be different from each other. Think of each piece of data read in from a sensor task as a data token. In the case of pipelined and multi-rate execution, multiple data tokens read in by the same sensor task in different execution periods may coexist in the system at the same time.

For illustration purposes, imagine that we attach a real-time clock that is reset to 0 with each data token coming in from the head of the pipeline, and mark this particular clock as being in use. Every time a data token goes through the entire pipeline and falls off the end of the task chain, we pick up the clock on that token, read off the value as the end-to-end delay suffered by that token, and mark the clock on that token as free and can be reused for a new token in the future. When the pipeline stalls at a task i, and the data token in the location before task i gets overwritten, We can also mark the clock of that data token as free. We can view a fresh data token read in from the environment as birth of that token, and token falling out of the end of the pipeline, or overwritten in the middle of the pipeline, as death of that token. A clock not currently in use is attached to each newborn token, and is marked in use. Upon death of a data token, either because it has gone through the entire pipeline, or it was over-written (killed prematurely) in the middle of the pipeline, we mark its clock as free and available for future use. It is important to note that when attaching a clock to a newborn token t_{new}, that clock must not be already in use by some other token t_{old} currently in the pipeline, for otherwise we will lose track of the time stamp of t_{old}.

If task execution is atomic and non-interleaving, then we need to keep track of clocks at the the inter-task buffer locations, as shown in Figure 1. Atomicity means that no other task can perform read or write operations in between a task's start and end. However, if task execution is non-atomic and interleaving, which means that it is possible for other tasks to perform read or write operations in between a task's start and end, then we need to keep track of clocks at

both the inter-task buffer locations and the task bodies between its start and end, as shown in Figure 2. In other words, we can view a task that executes non-atomically as two tasks, one read task and one write task, which execute atomically, and view the task body between its read and write operations as another data buffer.

Atomic execution is the case where tasks run on a single processor with non-preemptive scheduling, which effectively serializes the task execution. Non-atomic execution is the case where tasks run on a single processor with preemptive scheduling (pseudo concurrency), or where tasks run on a multi-processor platform (true concurrency).

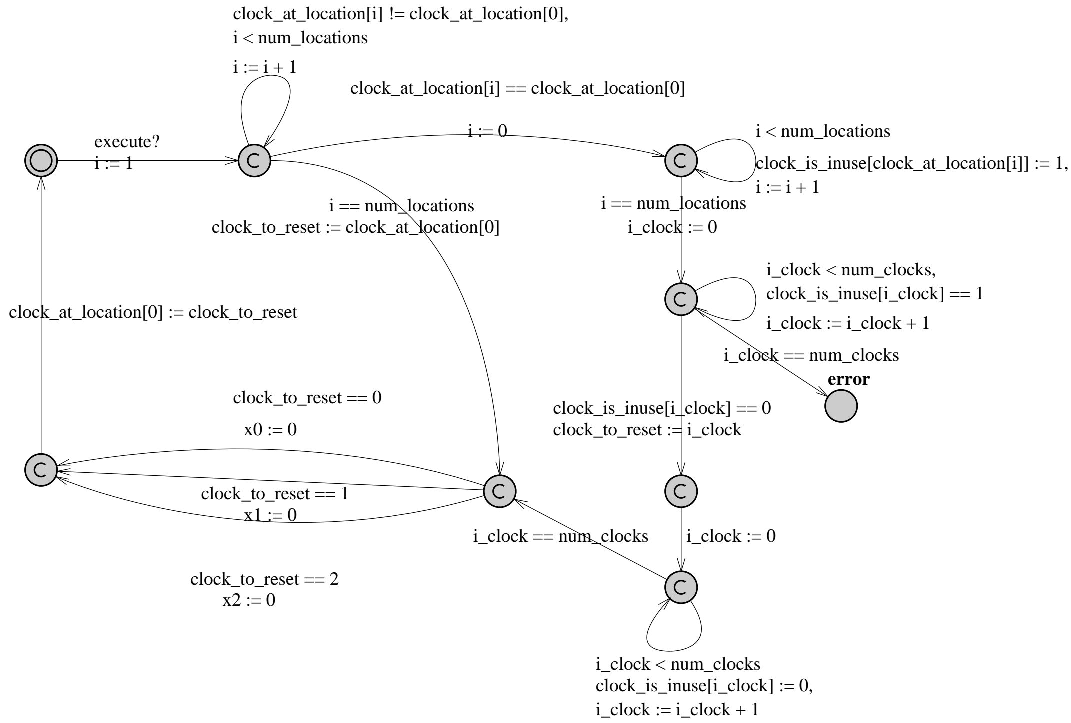


Fig. 3. TA template for tracking the timestamps of $location_0$.

The discussion above applies to a linear chain of tasks, but it can be easily extended to deal with a more general task graph with AND style synchronization. When a data token flows to an AND fork in the task graph, the token and its clock are replicated and sent along each of the forked paths. When several data tokens converge at an AND join in the task graph, all the tokens should be carrying the same clock that was replicated at the previous fork to ensure data consistency. Note that this behavior is not necessarily guaranteed. A certain branch of the AND fork may have a pipeline stall, so that the AND join point

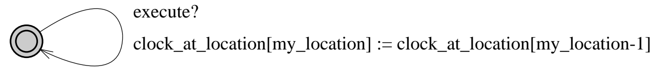
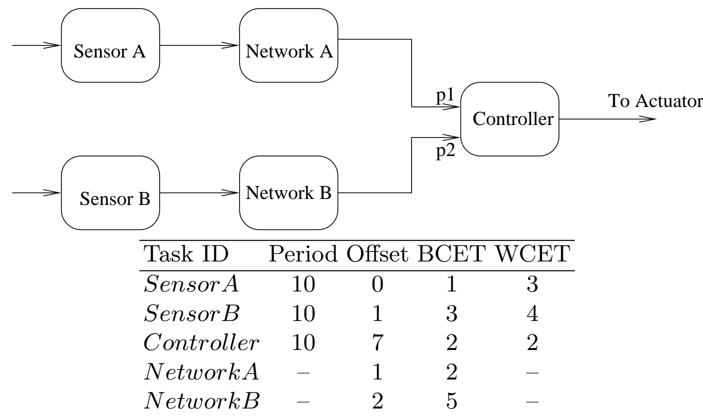


Fig. 4. TA template for tracking the timestamps of locations other than $location_0$.

may see different clocks from different incoming paths. This should be detected and flagged as a runtime error.

To instantiate these concepts within the modeling tool UPPAAL, Figure 3 shows a generic TA template for modeling the output buffer of the head of the task chain ($location_0$), and Figure 4 shows a generic TA template for modeling of buffers at locations other than $location_0$. The latter template is simple: the task simply copies the clock value at its input port to its output port. The task at the head of the chain needs to read in a fresh data token from the environment and attach to it a clock that is not currently in use. With these model templates, we can easily model larger systems by instantiating the model with different parameters.

3 An Application Example



Task ID	Period	Offset	BCET	WCET
$SensorA$	10	0	1	3
$SensorB$	10	1	3	4
$Controller$	10	7	2	2
$NetworkA$	–	1	2	–
$NetworkB$	–	2	5	–

Fig. 5. A system consisting of two distributed sensors and one central controller.

Consider the system in Figure 5, where two distributed sensors read data from the external environment and feed them into the controller, which processes

the sensor data and compute signals to the actuator. The network connections between sensors and controllers are modeled as separate tasks. We assume this is a distributed system, where each task has its own dedicated processor, so we do not need to address real-time scheduling issues caused by multiple tasks sharing one processor. We can view the system as being composed of 2 linear task-chains: Path A is from Sensor A to Network A to Controller, and Path B is from Sensor B to Network B to Controller. Task execution is non-atomic, hence each task chain corresponds to Figure 2. All tasks read input at startup and write output at finish. The table in Figure 5 shows the task timing parameters.

We have two system-level end-to-end timing constraints:

- *Freshness constraint*: the time interval between reading input data from either Sensor A or Sensor B to writing output data by the Controller must be less than 10ms;
- *Correlation constraint*: if input data d_1 read by Sensor A and input data d_2 read by Sensor B are used to produce output data d_3, then d_1 and d_2 must be sampled within 2ms of each other.

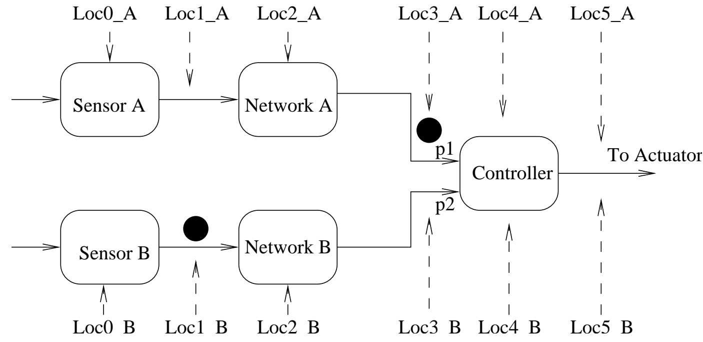


Fig. 6. Each black dot denotes one data token. The figure shows that the input data token read by Sensor A has gone through Network A and reached the input port $p1$ of the Controller, while the input data token read by Sensor B has not gone through Network B yet.

Figure 6 shows a snapshot during system execution. Figure 7 shows one possible execution trace for two consecutive execution periods. Sensor A reads input at time 0 and writes output at time 2ms. Network A is triggered by the completion event of Sensor A, and takes 1 time unit to deliver the message to input port $p1$ of the Controller. Similarly, Sensor B reads input at time 1ms and writes output at time 4ms. Network B is triggered by the completion event of Sensor B, and takes 2ms to deliver the message to input port $p2$ of the Controller. The Controller reads its input at time 7ms, and writes its output to the actuator at time 9ms. The next period is a repeat of the first. Therefore, the end-to-end delay of data tokens from Sensor A to the external actuator is 9ms, and from

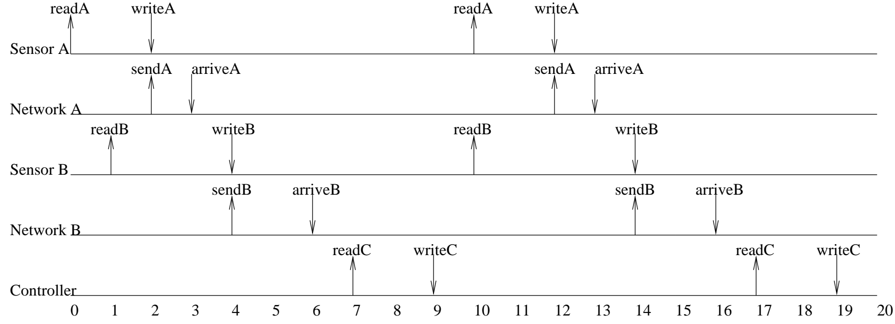


Fig. 7. One possible execution trace for the taskset specified in Figure 5.

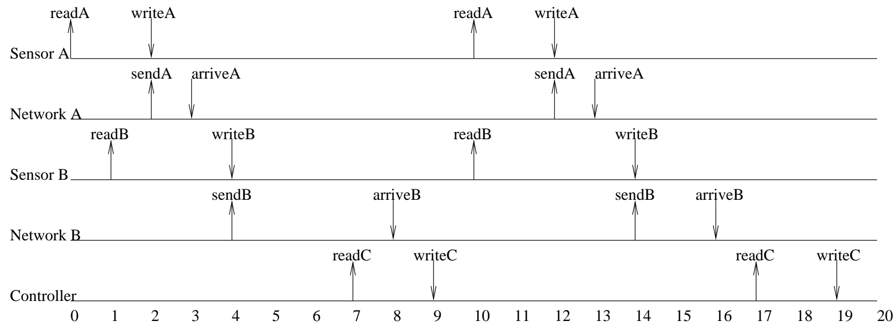


Fig. 8. Another possible execution trace for the taskset specified in Figure 5.

Sensor B to the external actuator is 8ms, both within the freshness constraint of 10ms. The correlation between data tokens at the 2 controller input ports is 1ms, i.e., the data token at $p1$ is sampled from Sensor A 1ms earlier than the data token at $p2$ from Sensor B. So the correlation constraint is also satisfied.

However, if Network B causes a slightly longer time delay of 4ms instead of 2ms, but still within the specification of [2,4]ms, then we have a different data token propagation scenario, as shown in Figure 8. Now the end-to-end delay for Sensor B's data token is 18ms, while the delay for Sensor A's data token remains at 9ms. So the freshness constraint is violated for Sensor B's data token. At the time of data reading of the Controller at time 17ms, the token at $p1$ from Sensor A is 7ms old, while the token at $p2$ from Sensor B is 16ms old. So there is a 9ms age difference between the two data tokens read by the Controller. This violates the correlation constraint, which states that the two input data tokens must be sampled within 2ms of each other. Intuitively, the data token read by Sensor A at time 0 was consumed by the Controller at time 7ms before the arrival of data token read by Sensor B at time 1ms, due to the long time delay caused by Network B. Therefore, the data token read by Sensor B at time 1ms is paired up with the data token read by Sensor A at time 10ms in the next execution cycle.

This caused the large age difference between the data tokens consumed by the Controller at time 17ms.

As this example indicates, a small additional delay at Network B can cause a relatively large difference for the end-to-end delay and input correlation. Therefore, it is important to analyze the system timing behavior carefully. For realistic systems with larger sizes, it is tedious and error-prone to perform the analysis by hand, therefore, it is desirable to apply automated techniques such as model-checking to discover timing anomalies by exhaustive state space exploration.

We have used UPPAAL to prove the following properties:

- The maximum end-to-end delay for sensor A data is 9ms.
- The maximum end-to-end delay for sensor B data is 18ms.
- the maximum age difference between the data tokens at the two input ports of the Controller is 9ms.

If we change the task parameters, so that Network B task causes a deterministic delay of 2ms instead of an interval of [2,4]ms, we can verify that the maximum age difference reduces to 1ms, and the maximum end-to-end delay of Sensor B data reduces to 8ms. This corresponds to avoiding the execution scenario in Figure 8.

4 Related Work

Wall *et al* [6] designed a TA model for end-to-end task graphs. However, the size of their model grows exponentially with the number of tasks, while the size of our model grows only linearly. We can model larger systems easily by instantiating the proposed modeling templates with different parameters. In general, to keep track of clocks at N buffer locations, the automaton in their approach consists of $N * N$ locations and $N * N * N$ edges. Furthermore, they only considered atomic task executions (non-preemptive scheduling on a single processor), and they used a task execution trace to drive the Time Stamp Tracker automaton. We take into account non-atomic task executions, and accurately model task execution dynamics, which enables exploration of the full system state space with model-checking.

Haveman *et al* [7] manually analyzed a task model that consists of 2 sensors and 1 monitor that merges the 2 sensor readings. They considered a general multi-rate execution scenario, where each task may execute at arbitrary periods that are not necessarily harmonically related, and derived equations for calculating various end-to-end timing properties. We can map their task model into the TA modeling framework discussed in this paper, and analyze it through model-checking, thus obviating the need for tedious and error-prone manual analysis.

There are other techniques and tools for model-checking for timed automata, e.g., Kronos [8], which provides slightly different definitions of timed automata. For example, UPPAAL's definition of TA adopts CCS-style [5] pair-wise synchronization between two automata, while Kronos' definition uses globally-shared

synchronization among all automata in the system. Our choice of the model-checker UPPAAL was motivated by several factors, including modeling convenience (pair-wise synchronization turns out to be convenient for our particular problem) and ease of use (UPPAAL provides a graphical user interface, while the other tools only provide textual interfaces). However, the main concepts of our approach are not restricted to any particular tool, but are generally applicable to other formalisms and model-checkers.

5 Conclusions

In this paper, we have explored application of model-checking to timing analysis of distributed end-to-end task graphs. We have developed generic modeling templates for a class of distributed task systems that can be used as input to the the model-checker UPPAAL in order to verify system-level end-to-end timing constraints. We have used a two-sensor-one-controller system to illustrate the complexity involved in end-to-end timing analysis, and applied our techniques to modeling and analysis of this system. Our approach is applicable to any real-time system that satisfies the assumptions stated in Section 2, i.e., tasks executing asynchronously and communicating through shared buffers.

References

1. R. Gerber, S. Hong, and M. Saksena, "Guaranteeing end-to-end timing constraints by calibrating intermediate processes," in *Proc. IEEE Real-Time Systems Symposium*, December 1994, pp. 209–213.
2. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL - a tool suite for automatic verification of real-time systems," in *Proc. Workshop on Verification and Control of Hybrid Systems*, October 1995, pp. 232–243.
3. H. Ben-Abdallah, D. Clarke, I. Lee, and O. Sokolsky, "Paragon: A paradigm for the specification, verification, and testing of real-time systems," in *Proc. IEEE Aerospace Conference*, Feb 1997, pp. 469–488.
4. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Times: a tool for schedulability analysis and code generation of real-time systems," in *Proc. International Workshop on Formal Modeling and Analysis of Timed Systems*, Sept. 2003.
5. R. Milner, *A Calculus of Communicating Systems, LNCS 92*. Springer-Verlag, 1980.
6. A. Wall, K. Sandstrom, J. Maki-Turja, C. Norstrom, and W. Yi, "Verifying temporal constraints on data in multi-rate transactions using timed automata," in *Proc. IEEE International Conference on Real-Time Computing Systems and Applications*, December 2000, pp. 263–270.
7. J. Haveman, "Transaction decomposition: refinement of timing constraints," in *Proc. South Pacific Conference on Formal Methods*, 1997.
8. S. Yovine, "Kronos: A verification tool for real-time systems," *Software Tools for Technology Transfer*, vol. 1, pp. 123–133, 1997.