Proceedings of the American Control Conference
Arlington, VA June 25-27, 2001

Motivating Hierarchical Run-Time Models in Measurement and Control Systems¶

Jie Liu†, Stanley Jefferson‡, and Edward A. Lee†

†Department of EECS
University of California, Berkeley
Berkeley, CA 94720, USA
{liuj, eal}@eecs.berkeley.edu

‡Agilent Laboratories
Agilent Technologies
Palo Alto, CA 94304, USA
stan_jefferson@labs.agilent.com

Abstract

Measurement and control systems are intrinsically distributed and real-time, as they contain sensor and actuator nodes that interact with the physical world directly. Embedded software in the computational nodes is responsible for timely reaction to sensor data, and for producing actuation. This paper reviews run-time computation models for this kind of real-time embedded software, from message semantics, message acquisition, and the dataflow/control flow perspectives. In general, dataflow centric models are natural for describing measurement and control algorithms and easy to use in distributed systems, but they lack mechanisms for controlling the execution order to fulfill timing constrains. Control-flow centric models are good at handling real-time requirements but are hard to distribute and sometimes hard to analyze. Although most practical run-time models to some extent support both dataflow and control flow, they are hardly universal. In complex applications, it is desirable to use different models in different parts of the system or under different modes of operation. Cleanly integrating multiple run-time models is a challenging task. In this paper, we motivate a hierarchical architecture for composing run-time models, based on the Ptolemy II component framework and models of computation. Unlike traditional real-time operating systems that provide only one flat layer of abstraction, the hierarchical architecture enhances flexibility, scalability, and reliability of MC systems by mixing and matching multiple run-time models in a disciplined way.

Keywords: *run-time models, hierarchical heterogeneity, real-time systems, embedded systems.*

1. Introduction

Measurement and control (MC) systems are distributed real-time computing systems that interact directly with the physical world. A typical MC system has sensor nodes, computing nodes, actuator nodes, and a communication subsystem that connects the nodes. As computation and communication resources get more economical and reliable, MC systems are becoming more complex and software-enabled. This increases the complexity of embedded software running on computational nodes. This kind of software usually carries tasks for controlling peripherals, collecting information, analyzing data, and producing reactions.

Run-time software for MC systems differs from its design-time counterpart and desktop software in many ways:

- **Real time**. Most run-time software for MC systems has constraints (either hard or soft) on the response time.
- **Concurrency**. MC systems directly interact with the physical world, so it is intrinsically concurrent. At the very least, the embedded system and its environment are operating concurrently.
- **Real I/O**. The interaction between a MC system and its environment are going through samples and events. How the state of the physical world is reflected in the computer is an important part of run-time models.

Following [15], we view a piece of software for MC systems as an aggregation of software components that interact with each other and with the environment by exchanging messages. These components could be processes, subroutines, or objects. A run-time model is a set of rules that governs the execution and interaction of these components. Run-time models are usually operating system (OS) concepts; for example, they may appear as part of real-time operating systems (RTOS) or virtual machines. They may also appear as application-specific schedulers or event dispatchers. As MC systems become networked, the concept of run-time models extends beyond single computer platforms.

Embedded real-time software has been an active research area for over 20 years. Many models have been proposed with different performance, predictability, and scalability [8] [10] [14] [18] [21]. However, as systems become more complex and large scale, no single model may fit for all applications. Nowadays, embedded real-time software is still a mixture of formal models and hand-tuned code, which make them hard to scale up, analyze and debug. It is also

¶. This work is supported by Agilent Technologies, and the Ptolemy project, which is supported by DARPA/ITO, the State of California MICRO program, and the following companies: Agilent, Cadence Design Systems, Hitachi, Hughes Space and Communications, Motorola, NEC, and Philips.

worthwhile to realize that different real-time models also impose different requests on hardware and system infrastructure. The designers must have a system-level understanding of the models and their implications.

This paper reviews some run-time models that can be used for MC software, in terms of their qualities of performance, predictability, and scalability. We argue that the traditional way of mapping all applications to a single run-time model is fragile and unscalable. We motivate a hierarchical run-time architecture based on the Ptolemy II component framework and models of computation [4]. By mixing and matching multiple run-time models, this architecture can ease the process of MC software development and improve run-time understandability, scalability, and reliability.

The rest of the paper is organized as follows. Section 2 discusses some perspectives on studying run-time models for MC systems, in terms of message semantics, data flow and control flow, and the notion of time. Section 3 gives some examples of run-time models. Section 4 motivates a hierarchical architecture for run-time embedded software. Section 5 discusses related work.

2. Perspectives on Run-Time Models

2.1. Messages in MS systems

MC systems have the physical world, in addition to users, as their environment. The environment generates messages that could be user commands, clock signals, alarms, as well as samples of physical variables like voltage, temperature, pressure, etc. A MC system responds to these messages and controls the behavior of the environment by producing output messages.

It is important to notice that the environment operates concurrently with the MC system. The MC system operates according to some run-time models, while the environment operates under its own laws of physics. In particular, in the physical world, time is continuous and flows at a constant rate. The environment never holds itself to wait for the next input. And, if an event occurs, the environment does not care whether the MC system is ready to receive it. Thus, the messages at the boundary of MC systems and their environment need to be carefully examined, especially when there is a mismatch on the execution rates.

2.1.1 Message semantics

Following Kopetz [13], we loosely classify two kinds of message semantics: *event* and *state*.

- Event semantics requires that the receiver of messages processes every event exactly once. The loss of a single event may lead to a misunderstanding between senders and receivers. If there is a mismatch between the production and consumption rates of events, a blocking mechanism or a queuing mechanism may be introduced to force synchronization. Event semantics can be further classified by whether the events carry time stamps, how they are ordered, and how to force synchronization.
- State semantics reflects the current state of the physical world. It is reasonable to only keep the most recent samples of physical states. The rate mismatch between senders and receivers can be solved by overwriting older data. State messages are usually seen in control-oriented real-time systems, where controllers only deal with the latest state of plants. Kopetz argues that in MC systems, state messages are far more frequent than event messages [13]. This argument may be overstated, but it reflects a fundamental difference between computing in MC systems and computing in traditional transaction-processing systems.

It is important to distinguish the message semantics from the general understanding of events and states. For example, in a data acquisition system, the data to be recorded are the "states" of a physical process. But if every sample is important, we may take the event semantics, and queue the samples if recording is slower than sampling.

2.1.2 Message acquisition styles

Traditionally, designers of MC systems think in terms of two styles of external message acquisition – *pushed* or *pulled*. A pushed message is actively sent by the sensor nodes, while a pulled message is queried by the computation nodes. Pushed messages usually generate interrupts. Although interrupts allow messages to be processed as soon as possible, they are one of the biggest sources of uncertainty in run-time software. Pulled messages does not generate interrupts, but they require extra CPU cycles for polling, and thus reduce the performance of the computational nodes.

As sensor nodes and communication systems become more intelligent, the boundary of push and pull styles is blurred. For example, a message pushed by a sensor can be locally cached by the communication layer of the computational nodes (as in TTP [14]) or middleware (as in Real-time CORBA [9]), and the run-time software will never notice that it is pushed. In addition, when there is a uniform notion of time in the system, sensors can time tag their readings, and the computational nodes do not have to poll the sensors very frequently. So, from the viewpoint of run-time software, the fundamental difference of message acquisition styles is whether they generate interrupts, and how the software responds to them.

2.2. Data Flow and Control Flow

It is helpful to look at real-time software models from the dataflow[1] and the control-flow perspectives. The dataflow

1. The term "dataflow" used here is close to but somewhat broader than that in dataflow programming languages.

perspective focuses on the data dependencies among components. A software component processes its input data and produces output data so that other components can use them. A component is eligible to execute when there are enough data to be processed. Most measurement and control algorithms, like filters, difference equations, compensators, and regulators can be conveniently specified in a dataflow way. In other words, in MC systems, the dataflow view of embedded software is close to the problem domain. Generally, the dataflow view only imposes a partial order on the execution of components. So, pure dataflow models have great flexibility on dealing with concurrency and are easy to scale up. On the flip side, the partial order of execution makes it harder for dataflow models to fulfill real-time constrains.

The control-flow perspective, like task scheduling and mode switching, focuses on controlling when a certain components be executed. The flow of control usually depends not only on the availability of data, but also on many other things, like the current state and time of the system, task priorities, and deadlines. It is thus straightforward to control the execution order so that the real-time requirements can be met, at least for simple systems. Control-flow centric models, although being capable of managing real-time properties, do not solve all MC software challenges. Some control-flow formalisms do not match the problem domain very well and the tight control on the execution order may make these models hard to reconfigure, hard to scale up and hard to use in distributed systems.

2.3. Notion of Time

Time, although being the most important concept in real-time systems, is not part of the computation in most models. This is partly because traditional computer sciences have systematically removed the notion of time from computer theories and partly because maintaining a synchronized time across platforms is difficult.

However, the development on hardware description languages (like VHDL and Verilog) and the study on discrete event systems [17] has shown that time can be an intrinsic part of computational models. Recent advances in timekeeping [1], time synchronization [14][25], and smart sensors and actuators [24] also show that the cost is relatively low to maintain a global notion of time in distributed MC systems. These technologies enable new infrastructure for real-time systems and time-based models [13] [22].

3. Run-Time Models

In this section, we give some examples of run-time models. This is by no means a complete list. The intent is to give a hint of the diversity of run-time models, their assumptions, and their quality of service.

- *Priority-driven multitasking*

Most real-time operating systems take priority-driven multitasking as their run-time model [2]. In this model, the software components are tasks, which are finite computations that process inputs and produce outputs. Individual tasks are usually highly dataflow oriented. Tasks may communicate through global variables, buffers, or queues. Tasks become eligible when they have enough inputs. When multiple tasks are eligible but the resources are limited, the run-time system chooses a task with the highest priority to execute.

There are various algorithms deal with when and how to assign priorities to tasks. The most famous ones are the rate monotonic scheduling and the earliest-deadline-first scheduling [18], which yield optimal CPU utilization under certain conditions. A basic assumption in these algorithms is that tasks can be arbitrarily preempted. In reality, this assumption may not always hold, and brute-force application of the algorithms may introduce undesirable phenomena, like priority inversion [20]. An alternative is nonpreemptive scheduling or cooperative multitasking. Although nonpreemptive scheduling may have weaker schedulability and lower CPU utilization than preemptive ones, they are more robust and easier to scale up. If tasks are short enough, nonpreemptive scheduling may provide close-to-optimal performance with small context switching overhead.

Priority-driven models are typically used within one computational node. The distribution of such models requires the system to be coherent [21], which is stronger than prioritized event dispatching and a system-wise understanding of priorities. These requirements may be hard to maintain across heterogeneous platforms.

- *Time-triggered architectures*

Time-triggered architectures [14], assume the state semantics of messages, and control the execution of components solely based on elapsed time. A distributed time-triggered architecture needs a time-division-multiple-access (TDMA) communication protocol and (roughly) synchronized clocks across nodes, which may require additional hardware support. In these models, time is segmented into frames. Computation and communication are both triggered by the start of a time frame, and they must finish by the end of the time frame. Since time triggered architectures directly address time in the computation, they can guarantee to deliver hard real-time performance. On the flip side, they usually have low CPU utilization and are too rigid for soft real-time applications.

- *State machines*

At run time, state machine models can be used in two levels, within a component or across components. Within a component, state machines are sequential and apply precise control

to component states and transitions among them. State machines are amenable to in-depth analysis and formal verification, which make them desirable for safety critical systems. Many design-time models, like Statechart [7], Esterel, Signal, and Lustre [6], synthesize their components into state machines to be executed.

State machines can also be used across components to coordinate other models. The typical use is to specify operation modes and sequences among components. The Statechart formalism is an example of using high-level state machines to coordinate low-level state machines. The "*charts" (pronounced star-charts) model [5] extends the coordination to a huge variety of concurrent models. Hierarchically combining state machines with concurrent models makes state machines concurrent and helps to prevent the explosion of the number of states in complex systems. But when and how to make state transitions remain challenging issues. Carelessly implemented concurrent models and randomly made transitions may lead the system to non-quiescent states, which should be avoided at all cost.

- *Asynchronous message passing*

In asynchronous message passing, components are processes communicate by sending messages through FIFO queues. The sender of a message need not wait for the receiver to be ready to receive the message. There are several variants of this model, for example, various dataflow models, but they share the same property that components are loosely coupled. Thus, these models are easy to distribute. Most asynchronous message passing mechanisms impose few constrains on the execution order of components, which makes the performance of such models highly unpredictable.

Synchronous dataflow (SDF) [16] is a particularly restricted special case of asynchronous message passing with extremely useful properties. In SDF, whenever an component executes, it consumes a fixed amount of input data and produces a fixed amount of outputs. This property makes questions like deadlock and boundedness decidable [12]. For a consistent SDF model, a schedule can be computed, such that the components do not have to test for sufficient data before execution. So, for computation that has a fixed structure, SDF is very efficient and predictable.

- *Publish and subscribe*

Publish and subscribe (P/S) models extent event dispatching mechanisms to distributed systems. In P/S models, event channels [11], also known as a persistent object spaces, mediate the communication between senders and receivers. Since no direct channel needs to be established between senders and receivers, P/S models are good for managing communicating peers that join and leave a federation. Prioritized P/S models dispatch events according to their priorities [9], so it gives more fine-grained control on the order of transmitting events.

- *Time-Synced event driven*

When there is a highly accurate synchronized time across a distributed system, the nodes can coordinate their operation with respect to time. In particular, sensor nodes can attach time stamps to their readings and these time stamps will make sense at the computational nodes. Computational nodes can send time-stamped outputs to actuators and the actuator will perform the output at the right time without further involvement from computational nodes. This mechanism frees the computational nodes from time tracking and allows them to process events faster than real time, and thus called faster-than-real-time computation [22].

4. Hierarchical Run-Time Models

Complex MC systems usually have multidimensional quality of service requirements. Some parts of a system may require hard real-time guarantees, some parts may require high utilization of resources, and some other parts may require immediate response to spontaneous events. These requirements may also change with respect to time and operational modes. Ad-hoc integration of different run-time models may bring emergent behaviors and destroy the analyzability and reliability of a system. We propose a component-based architecture to hierarchically integrate multiple run-time models. This architecture is based on the Ptolemy II component framework and models of computation.

As in Ptolemy II, a basic software component is an *actor*. Actors have well-defined communication points, which are called *ports*. How a port is implemented depends on the run-time model that the actor is in. An actor, when executed, performs a finite atomic execution and reaches a quiescent state. Such execution is called a *precise reaction*. A run-time model is implemented as a *director*, which defines the communication style (i.e. the ports) and the execution order of the actors under its control. For example, a priority-driven multitasking director will manage the priorities of actors and an event queue that dispatches events according to their destination actor's priority. A time-triggered architecture director may provide state semantics with double buffering for communication and use synchronized timers to schedule computation and communication. A publish/subscribe director will realize an event channel and dispatch events over network.

An aggregation of actors, together with their director is called a *composite actor*. With careful design of directors, a composite actor can work exactly like an atomic actor. That is, a composite actor will have well-defined communication points and precise reactions. Such a composite actor can be integrated with other actors under the control of a different director. In this fashion, multiple models can be hier-

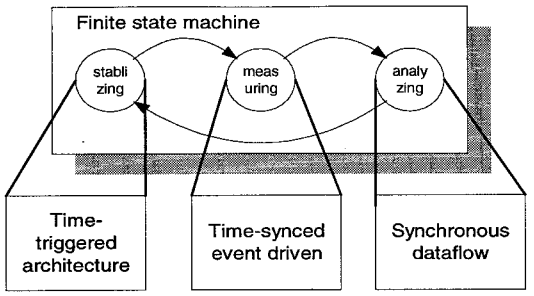

Figure 1. Hierarchical models for a test and measurement system.

archically integrated, and within each level there is a well-defined model. This architecture localizes the run-time models only to the components that need them and preserves their properties within that range.

For example, consider a typical test and measurement application, where the plant to be tested is first stabilized and driven to a neutral operation point, then a sequence of stimuli (a.k.a. test vectors) is injected and the response of the system is measured, and finally the data collected are processed to extract testing results. The run-time system may work in the following hierarchical way, as illustrated in Figure 1. The top level is a finite state machine, which controls a sequence of operations: stabilizing, measuring, and analyzing. In the stabilizing state, a periodically sampled feedback control is used to drive the plant to the neutral state. A time triggered architecture guarantees the timing of the control. In the measuring state, a time-synced event driven model can be used to apply a sequence of test vectors to the plant and start the data collection process accordingly. In the analyzing state, the system is off-line. Since the analysis algorithms, like filtering and spectrum analysis, have static structures, SDF provides high computational throughput. Each operational mode has precise reaction, and the mode switching points are well-defined.

Consider another example of a fault tolerant control system for air-vehicles, as shown in Figure 2. Sensors, computers, and actuators can be connected by a publish and subscribe event channel. Multiple tasks, for example a high-priority

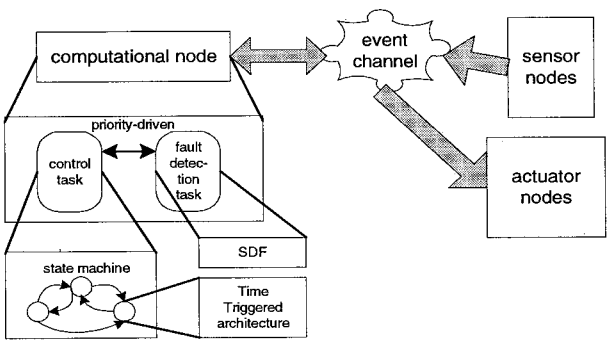

Figure 2. Hierarchical models for a fault-tolerant control system.

feedback control task and a low-priority fault detection task, run on the computer. Depending on the operational status, the feedback control task may have multiple modes, each taking care of a region of flight trajectory. In particular, there is a safety protection mode, and alarms from fault detection task will trigger the controller to go into that state. Within each state, a time-triggered architecture can be used to support the implementation of sampled-data control laws. The fault detection task is soft real-time and maintains its own data space. Thus, it can be preempted by the control task at any time.

5. Discussion

Systematically integrating multiple models is crucial to design large-scale distributed real-time systems. Many active research projects address this issue and influence our architecture. For example, [3] gives conditions and architectures for distributing synchronous languages. By posing slight constrains on components and communication protocols, asynchronous event passing can be used to coordinate synchronous execution of components and the composition maintains the synchronous semantics. This leads to a globally asynchronous and locally synchronous (GALS) architecture for reactive systems. Giotto [10] integrates multirate time-triggered architecture with finite state machines. The open control platform (OCP) [19] integrates a prioritized publish and subscribe model for event dispatching with priority-driven multitasking on single nodes. But most of these projects only integrate two models and assume a fixed containment relation between them.

It is important to understand the synergy and distinction between a heterogeneous design environment and a run-time system. Ptolemy II is a heterogeneous modeling and design environment, which is the root of this work. Design-time environments emphasize the understandability of models, syntax and semantics checking (like type systems), and component polymorphism, while run-time systems emphasize physical interface, performance, and footprint. There are certain design-time models that are unsuitable for run-time systems. Those models may be nondeterministic and possibly deadlock, may make extreme assumptions, or may only be useful for modeling physical environment but not embedded software.

Code generation is a migration path from certain design-time models to run-time models [23]. A typical code generation process assumes a flat operating system support and generates a stand-alone program that is then compiled into an application. The wide variety and irregularity of MC system platforms and operating systems make the code generation process burdensome and unportable. We argue that having a hierarchical run-time system will greatly ease code generation and improve the quality of final applications. A run-time system can use platform dependent hardware sup-

port, instruction set and I/O to provide high quality services to applications. In addition, there are certain operations, like preemption, can only be achieved by OS-level run-time systems, but not easily by applications.

Notice that another view of integrating heterogeneous run-time models is to show that they can all be implemented by a grand unified model. This is the generalization of the traditional operating system view that a flat layer of abstraction will fit for all applications. For example, it is possible to claim that all the models in section 3 could be implemented by a time-synced distributed priority-driven model. There are at least two disadvantages in such viewpoint.

1. A grand unified model usually provides little analysability. Undisciplined mixing of arbitrary features makes applications fragile.
2. An applications usually does not need all the features provided by the grand unified model. Packaging and integrating only the necessary run-time support will help improve performance and reduce footprint.

6. Conclusion

Noticing a wide variety of run-time models for distributed measurement and control systems, their assumptions, and quality of service, this paper motivates a hierarchical architecture to integrate multiple models. Unlike a traditional RTOS, which provide only one flat layer of run-time models, this architecture keeps a clean model at each level and uses hierarchical composition to mix and match heterogeneity.

References

[1] D.W. Allan, N. Ashby, and C.C. Hodge, *The Science of Timekeeping*, Hewlett-Packard Application Note 1289.

[2] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-Vincentelli, "Scheduling for Embedded Real-Time Systems," *IEEE Design and Test of Computers*, Jan.-March 1998, pp. 71-82.

[3] A. Benveniste, B. Caillaud, and P. Le Guernic, "Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation," *Information and Computation*, 163 (2000), pp. 125-171.

[4] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong, *Ptolemy II: Heterogeneous Modeling and Design in Java*, technical report, UCB/ERL No. M99/44, University of California, Berkeley, July 1999.

[5] A. Girault, B. Lee, and E.A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 18, No. 6, June 1999.

[6] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers. 1993.

[7] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, 1987, pp. 231-284.

[8] D. Harel and A. Pnueli, "On the Development of Reactive Systems," *Logic and Models for Verification and Specification of Concurrent Systems*, Springer Verlag, 1985

[9] T.H. Harrison, D.L. Levine, and D.C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," *Proceedings of OOPSLA '97*, Atlanta, GA, October 1997, ACM.

[10]T.A. Henzinger, B. Horowitz, and C.M. Kirsch, *Giotto: A Time-triggered Language for Embedded Programming*, Technical Report, University of California, Berkeley, UCB//CSD-00-1121, 2000.

[11]K. Juvva and R. Rajkumar, "A Middleware Service for Real-Time Push-Pull Communications," *Proceedings of IEEE Workshop on Dependable Real-Time E-Commerce Systems* (DARE'98) June 1998.

[12]R.M. Karp and R.E. Miller, "Properties of a Model for Parallel Computation: Determinacy, Termination, Queuing," *SIAM Journal*, Vol. 14, pp. 1390-1441.

[13]H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997.

[14]H. Kopetz and G. Gruensteidl, "TTP - A Protocol for Fault-Tolerant Real-Time Systems," *Computer*, Vol. 27, No. 1, Jan. 1994, pp. 14-23.

[15]E.A. Lee, *Embedded Software - An Agenda for Research*, UCB ERL Memorandum M99/63, University of California, Berkeley, ERL

[16]E.A. Lee and D.G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, Sept. 1987.

[17]E.A. Lee, "Modeling Concurrent Real-time Processes Using Discrete Events," *Annals of Software Engineering*, Special Volume on Real-Time Software Engineering, vol. 7 (1999), p.25-45.

[18]C. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *ACM Journal*, Jan. 1973, Vol. 20, No. 1, pp. 44-61.

[19]J. Paunicka, B. Mendel, and D. Corman, "The OCP: An Open Middleware Solution for Embedded Systems," to appear in 2001 *American Control Conference*, Arlington, VA, June 2001.

[20]R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, Kluwer Academic Publishers, 1991.

[21]L. Sha, R. Rajkumar, and S.S. Sathaye, "Generalized rate-monotonic scheduling theory: a framework for developing real-time systems," *Proceedings of the IEEE*, vol.82, (no.1), Jan. 1994. p.68-82.

[22]D. Tennenhouse, "Proactive Computing," *Communication of the ACM*, Vol. 43, No. 5, May 2000, pp. 43-50.

[23]J. Tsay, C. Hylands and E.A. Lee, "A Code Generation Framework for Java Component-Based Designs," 2000 *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, (CASES'00), November, 2000, San Jose, CA

[24]S.P. Woods, B. Hamilton, and J.C. Eidson, "The Advantage of Implementing Synchronized Clocks in Distributed Measurement and Control Systems," *Sensors Expo.*, May 2000, Anaheim, CA

[25]*Draft Standard for a Precision Time Protocol* (PTP), v0.15, prepared by the Systems and Solutions Laboratory, Agilent Laboratories, April, 2000.