Model-Based Autosynthesis of Time-Triggered Buffers for Event-Based Middleware Systems [*]

Jonathan Sprinkle
University of Arizona
sprinkle@ECE.Arizona.Edu

Brandon Eames
Utah State University
beames@engineering.usu.edu

ABSTRACT
Application developers utilizing event-based middleware have sought to leverage domain-specific modeling for the advantages of intuitive specification, code synthesis, and support for design evolution, among other benefits. For cyber-physical systems, the use of event-based middleware may result, for some applications, in a need for additional time-based blocks that were not initially considered during system design. An advantage of domain-specific modeling is the ability to refactor an application to include time-triggered, event-based schedulers. In this paper we present an application in need of such modification, and discuss how these additional blocks must be synthesized in order to conform to the input/output constraints of the existing diagram.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Prog. Environments—*Integrated environments*

Keywords
Metamodeling, software synthesis, graph rewriting

1. INTRODUCTION
Cyber-physical systems involve algorithms and design techniques from the disciplines of control, real-time systems, robotics, software, communications, and many other application domains. Experience spanning multiple disciplines is required when developing these kinds of systems in order to manage the various subtleties of each domain, in addition to the subtleties of their integration.

In existing implementations of today's cyber-physical systems, designers commonly employ one of many discrete event-based computational models [25, 6, 11, 26]. In these models, components execute based on the exchange of tokens with other discrete event components. On execution, components acquire input tokens from their associated input queues, perform computation and submit their results to any consumers via output queues. A common event-based execution approach is realized through the application of event-based middleware, where (potentially) distributed components communicate through message passing in order to exchange data. The receipt of data typically triggers component execution.

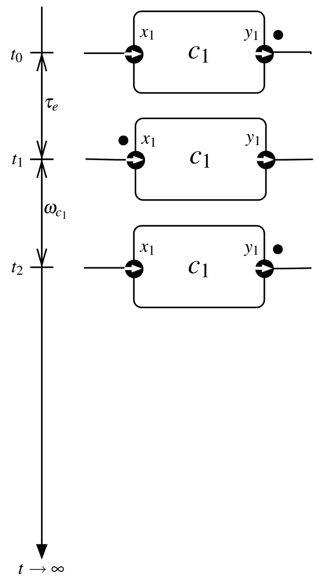
Generally, components for such systems are developed by algorithm experts who understand well their computational behavior. Occasionally when a system is composed from components whose execution triggering rules are determined in an ad hoc manner by each developer, the behavior of the system is emergent in nature, as opposed to being engineered by design. A more principled system design could consider the semantics of the composition of the components (as discussed in [7]). However, such integration is difficult to enforce in a programming styleguide since the semantics are at a higher level.

In this paper, we discuss how to enforce a time-triggered, as opposed to purely event driven, behavior through the insertion of data buffers whose contents are read and released on the receipt of time-triggered tokens. We provide some examples of how existing domain-specific models of event-based middleware can be rewritten in order to produce new component graphs that now implement this kind of scalable behavior.

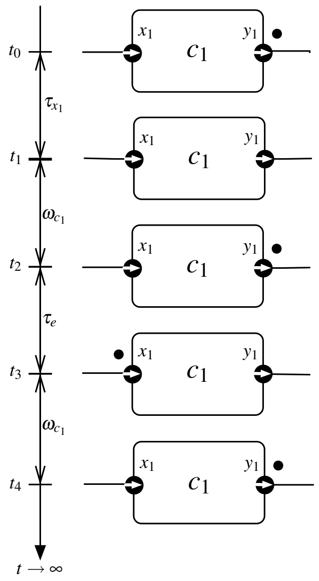
2. BACKGROUND
2.1 Middleware
The growth in the number of middleware technologies, and their widespread adoption in large scale system design is a testament to their utility in mitigating low-level programming complexity in distributed system development. CORBA offers a middleware platform for supporting distributed computing. Real-Time Object Request Brokers (ORBS) [19] have been developed, e.g. TAO [22], which integrate operating system services and network protocols to offer predictable quality of service, including real-time or near real time response. TAO allows distributed applications to be specified as a set of interacting components. The middleware services manage data communication between components, including marshalling and demarshalling, allowing components to be written location-unaware. A variety of competing middleware technologies and platforms have been developed for supporting component-based distributed computing, including Ice [8], Enterprise Java Beans (EJB) [15], the Microsoft Component Object Model (COM) [3] and .NET framework [17], and Java RMI [24].

A key goal with middleware is the development of Quality of Service guarantees (QoS) for supporting application execution. Certain metrics are critical to distributed application development, e.g. bandwidth, latency and jitter. Different studies have been conducted to evaluate and improve the

[*]This work supported by the Air Force Research Labs, under award #FA8750-08-1-0024, titled "MultiCore Hardware Experiments in Software Producibility."



(a) Event-Based Execution, $\tau_e \leq \tau_{x_1}$



(b) Timeout Execution, $\tau_e > \tau_{x_1}$

Figure 1: Alternative execution of a component with timeout on a blocking read. (a) A token is received before, or at, the timeout of the blocking read. (b) No token is received by time t_1, so a timeout event occurs, and the output token is based on the previous input to the component. Receipt of later events occurs as in (a).

ability of middleware to offer predictable quality of service [21, 20]

2.2 Publish/Subscribe Methods

Publish/Subscribe is a common model of communication used for data interchange between components. A component c_1 will subscribe to another component's production of a particular data value, using services offered by the middleware. Then, the middleware will ensure that all subscribers receive the value once it is produced. This can be considered to be closely related to the producer/consumer model, however the producer/consumer model typically only involves a single consumer for a given producer. The publish/subscribe model offers explicit support for a controlled broadcast communication.

Several middleware technologies support variants of the publish/subscribe communication model, including Ice [8] and DDS [16]. Further, different approaches for implementing the model exist, including having the consumer task poll the producer for data availability, or having the producer perform a remote invocation on the consumer when data is available.

2.3 Time-Triggered Methods

Different techniques and infrastructures have been developed to support distributed/embedded computing which executes cognizant of time. Giotto [9, 10] offers a framework for capturing and executing time triggered software. Applications are captured as a set of interacting tasks, each of which is assigned to a mode with an associated execution period. Scheduling and inter-task communications are managed by Giotto's runtime infrastructure, called the E-machine.

Farcas et al. [5] have developed a component model to support the development of distributed real time systems. Components are decoupled from the platform and from each other, both temporally and functionally. The component model is based on the logical execution time (LET) model introduced with Giotto. Auerbach et al. [1] describe a Java-based approach for scheduling real-time control tasks on a quad-rotor model helicopter by making use of exotasks and the LET model to guarantee deterministic execution.

The Time Triggered Architecture (TTA) [12] and Time Triggered Protocol (TTP) [13] have been developed to support the time-based execution of components in hard real-time systems. Components are allowed to access shared communication hardware based only on the clock, rather than need. Design time scheduling determines *a priori* the allocation of access windows to shared busses. The goal of time triggered execution is to isolate the impact of component or subsystem failures on the system as a whole.

3. DOMAIN SEMANTICS

The domain of event-driven component-based systems provides significant freedom in the implementation of execution semantics. This freedom, in part, motivates the need to constrain execution across implementation platforms.

Our purpose in this section is to underscore the *semantic* reasons for which components specified in our language could execute differently on machines with different network latencies, or processing power. This understanding motivates why we undertake this transformation process, and additionally provides the semantics necessary for the components which we synthesize using the model transformations in the next section. We provide here several examples that demonstrate the subtle behavior anomalies that are a result of a purely event-driven model of computation. We continue with a concise description of the execution semantics of a time trigger component, which will be integrated into the system to eliminate these anomalies.

3.1 Single Input/Single Output

Consider a component, c_1, with a single input, x_1, and single output, y_1. We use subscripts to denote the component, input, and output index, respectively, in order to provide consistent labeling for multi-component, multi-input/output systems that we will describe in future sections. The value y_1 is obtained as the output of the functional behavior of the component, which may also be written in difference equation form as $y_1(k+1) = f(x_1(k))$, demonstrating the discrete notion of the software component, and our ability to encode y, x as signals in time (specifically, discrete time)[1]. The execution time of c_1 is not necessarily a constant, though we represent it by the variable ω_{c_1} for simplicity of specification, and leave open the potential that ω_{c_1} may be a random variable. In hard real-time systems ω_{c_1} (the execution time of the function) can be determined through worst-case execution time (WCET) analysis [18], and perhaps even be validated at the hardware level [4]. This component also has associated with it a time, τ_{x_1}, which is a timeout constant.

An example execution based on events received is shown in Figure 1a. At t_0, the logical beginning of this cycle, the system has just produced an output on y_1. At time $t_1 = t_0 + \tau_e$, an event occurs where a token is received on the input port, x_1. The component executes, and at time $t = t_2 = t_1 + \omega_{c_1}$ a token is produced on the output port, y_1, i.e., $y_1(t_2) = f(x_1(t_1))$. This execution is valid for any system execution where $\tau_e \leq \tau_{x_1}$, or where the component does not utilize a timeout.

In order to see how tokens flow in an execution where timeout is a factor, examine Figure 1b. At t_0, the system has just produced some token. At some time, $t_1 = t_0 + \tau_{x_1}$, a timeout event occurs, so the output port produces another token, namely $y_1(t_2) = y_1(t_0)$. That is to say, the output at $t_0 + \tau_{x_1} + \omega_{c_1}$ is equivalent to that produced at t_0. This equivalency ignores metadata such as send/receive timestamps, packet size, etc. Another remark is that the duplicative behavior of this component need not be the only behavior in terms of timeout. Many components may opt to produce a special timeout token, or no token at all, in the case of timeout. This is also a valid option, though in our case we aim to address systems that integrate tightly with physical systems, and inaction may be inappropriate in this domain.

An activity diagram that considers each of the cases presented in Figure 1a and Figure 1b is shown in Figure 2. In this diagram, the mutually exclusive case of receiving data, or timing out, is clearly shown. However, designs such as this (if already fielded) will satisfy these observations: (1) algorithms to discard "stale" data must already exist; and (2) timeouts, τ_{x_i}, will be appropriate for the execution time ω_{c_i} of this component, as well as the frequency of execution

[1]Of course, the internal state of an object can affect this outcome, but externally the interface is as presented.

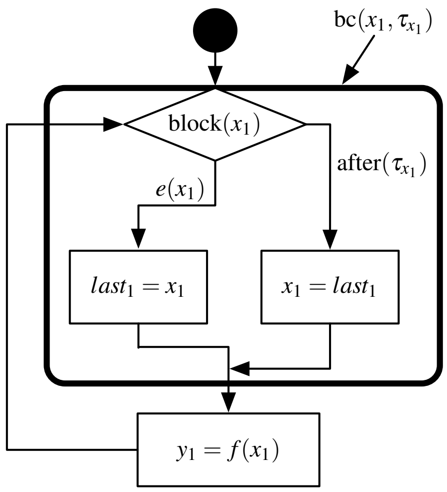


Figure 2: Activity Diagram that incorporates behaviors seen in Figure 1a and Figure 1b. The notation $e(x_1)$ indicates that an event was received indicating new data on input x_1; these events are cached if not being blocked upon, but the number of data values cached is application dependent.

for all providing components.

For brevity, we mention that the approach, and issues, for SISO systems can be generalized for MIMO systems. Due to the scope of the workshop, we leave this discussion to future papers in the domain, rather than the language elements of our work.

3.2 Trigger Generator

We provide the semantics for a particular component called a trigger generator. This component produces, at specific times or rates, a special token whose data is the time at which the token was generated. The default internal structure of the trigger generator component is a single output port. Tokens are produced on the port according to some internal parameters specified for the component, which include wait time, w, start modulus, m, and period, T. Usually, either w or m is specified, and once that time arrives a token is produced, and another token is produced every T seconds.

As a matter of implementation, empirical results from our work shows that using a `pthreads` [14] enabled operating system[2] (but not a real-time OS) results in a variance in expected time generation of approximately 2-3 milliseconds. On a real-time OS[3], the variance is less than 1 ms.

3.3 Buffer Semantics

A buffer component, C_b provides an integer number of outputs, j, with inputs $k = j + 1$. The j output ports match to the j inputs of some existing component being buffered, C_a. Values, when received by an input, are queued by C_b. One particular input, the time trigger input, subscribes to the single output port of a trigger generator component. When

[2]Linux flavors Kubuntu and Gentoo were used.
[3]QNX was used for the RTOS.

a token is received on the time trigger port, the queued data values are sent to the output ports such that they can be received by C_a.

In future work, we may provide a special semantics for buffers where the most recent value received (only) on each input port is passed to the output port. Our current semantics requires that if more than one value (or no value at all) is received by the buffer between triggers, then C_a is responsible for determining whether to use all, or only the most recent, values.

4. TRANSFORMATION AND EXAMPLE

Given the advantages of a system whose timing characteristics are explicitly expressed, we describe now a transformation from a purely event-driven model to one with time-triggered execution. This transformation modifies an existing graph, and permits existing components to execute with no behavioral changes: the only change to the system is the topological rewrite, and the insertion of new buffered components along with their time-based triggers. Our methodology is as follows: (1) examine an existing component interconnection graph; (2) insert, before each component of the graph, a time-triggered buffer; (3) insert, after each component of the graph, a time-triggered buffer; and (4) insert, somewhere in the graph, a timed event-generator for each buffer, which sends events at the appropriate time The rewriting rules for this methodology are trivial, when specified using the GReAT rewriting language [2]. In this section, we provide a subset of the transformations required.

4.1 Domain-specific Modeling Language

Our systems are defined using a domain-specific language, capable of synthesizing experiments based on component interconnections. This work is explained thoroughly in [23]. The metamodel is shown in Figure 3.

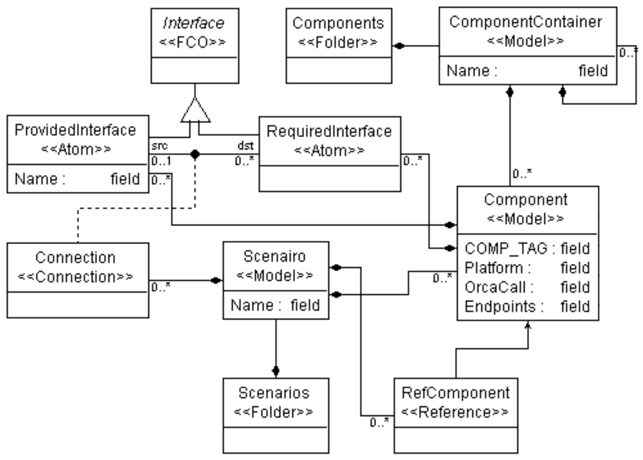


Figure 3: Relevant subset of the metamodel of our DSML (a screen capture from the GME tool).

Our system is built mainly of `Component` objects, connected to one another through provided and required `Interface` objects. The language provides several constraints (outside the scope of this paper) to prevent ill-formed models. We use this metamodel as the source *and* destination metamodel in the GReAT transformations.

4.2 Transformation Rules

The first task of the rewriting algorithm is to create new buffer components, whose job it is to implement the semantics described in Section 3. The precise semantics are generally decided for the entire graph, not piecemeal, and can be a value selected when the graph transformation is executed. For brevity, we have provided an abbreviated algorithm, that only inserts buffers for the *input* of each graph component. The algorithm to insert for component outputs is similar. An overall description of the rewrite is summarized by the rules shown in Figure 4.

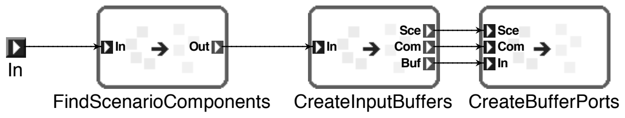


Figure 4: The order of rule execution for rewriting.

Buffers are added as `Component` objects, and the executable for this object can be generically synthesized based on the number of input/output ports, and the semantics chosen. We leave this detailed discussion to future papers, and concentrate instead on their insertion based on context. In Figure 5 the rule details are shown.

The rule can be read as follows: for each `Component` that contains a `RequiredInterface` object, create a new `Input-Buffer:Component`, with a `RequiredInterface` to accept time triggers, that will become its input buffer. Create also a new `Trigger:Component` with a `ProvidedInterface` that will provide time triggered events, and connect that provided port to the required trigger in the new buffer. Various objects are renamed for clarity in the final model.

We must also replace existing connections between two `Component` objects by routing those connections between the new `InputBuffer:Compoment` that we created above. A similar rule (not shown for brevity) removes the existing `Connection` and creates two new ones, such that the `Input-Buffer:Component` maps the data through to the `Component`.

Other rules not shown insert the necessary configuration items for each trigger component, as discussed in Section 3. These include the frequency of execution (based on the WCET of the component), and the time modulus at which to start running.

Executing all of the rules gives the transformation result shown in Figure 6.

5. ANALYSIS AND DISCUSSION

What role does Domain-Specific Modeling play in this application of transformations? In fact, why is modeling useful as a design concept here, rather than just using "old-fashioned" programming to solve the problem? What is the particular advantage that model transformations give to enable this migration from event-based to time-based behaviors?

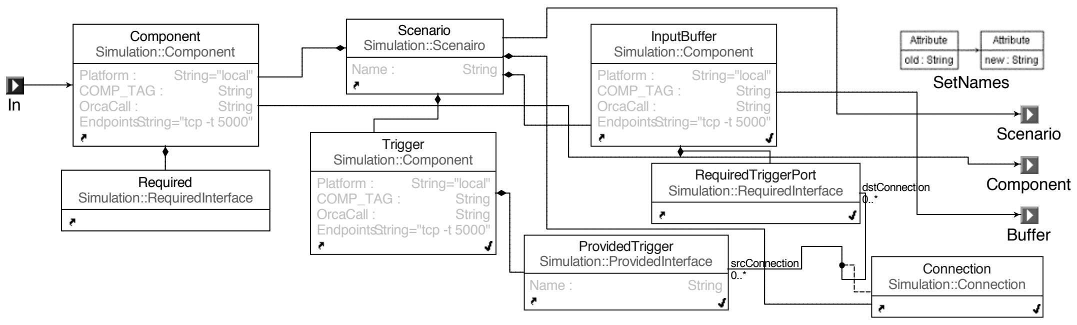
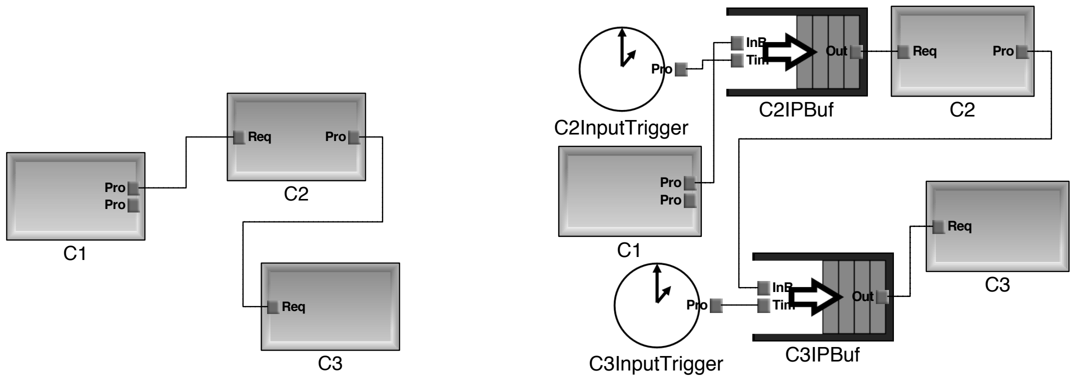


Figure 5: GReAT rule to create buffer components for triggered input reading based on time-triggered events. Note that the trigger scheduler is generated at the same step.



(a) Original model. (b) Transformed model.

Figure 6: (a) An original model, with only event-based triggering. (b) The result of transforming (a) with the rules discussed in Section 4.2.

With response to the relation to DSM, the application requires a domain-specific encoding of existing solution. The language we designed places particular emphasis on event-based behaviors, and the simple interfaces defined between components of production and consumption enable the direction of data to be well understood. Thus, such a language is able to capture existing systems, *and run them* [23]. This is an important distinction: a design that concentrated mainly on how the *new* system, with its new semantics, should be represented would be unable to validate that an existing model accurately represents the system as is.

So, why not use just clever programming techniques to migrate systems to the new semantics? Certainly this approach could be taken, but at the usual risk of requiring experts in the system design to become experts in new semantics, and new techniques for execution. Our approach permits existing system components to *continue to work using their old semantics*. Thus, not one line of code needs to be changed in the existing software to conform to the new time-triggered behaviors.

Additionally, system experts know how their systems are currently organized, and implemented. These experts can take our language and represent the models both as they are, and (perhaps) as they *should* be in the future. The model transformation approach to creating the new buffers is advantageous in that it reuses the investment in the encoding of the system into our DSML.

The model transformation approach (through GReAT) reuses the metamodel-based specification of the DSM in a way that language designers can discuss how types are used, and new components are generated, with the system experts. We again point out that this permits reuse of existing codebases. Approaches that do not use the strong typing and constraint-based organization that DSMLs provide to the end user run the risk that some corner cases may not be covered, or that assumptions made by the migration software are invalid according to the metamodel.

6. CONCLUSION AND FUTURE WORK

We have described how event-driven component-based systems can experience differences of execution based on subtle timing changes. We presented the notion of inserting time-triggered buffers as a way to reuse existing component code, while increasing the timing accuracy of component execution. We showed how, with an existing DSML to model such component-based systems, we can use model transformation techniques to enforce our time-triggered semantics on an existing model. We provided a semantics for these buffers, as well as the time triggers that control how long buffers hold their data tokens.

Our future work includes various autogeneration of the buffer code for various component-based middleware frameworks. We can also become more sophisticated in the synthesis of time trigger components, to attempt to consolidate the necessary scheduling into a single component, rather than a distributed set of components, thus increasing the scalability of the system without increasing the number of components in the system linearly.

Acknowledgments

This work is supported by the Air Force Research Laboratory, under award #FA8750-08-1-0024, titled "MultiCore Hardware Experiments in Software Producibility."

7. REFERENCES

[1] J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. T. Rajan, H. Roeck, and R. Trummer. Java takes flight: time-portable real-time programming with exotasks. *SIGPLAN Not.*, 42(7):51–62, 2007.

[2] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. The graph rewriting and transformation language: GReAT. *Electronic Communications of the EASST*, 1, 2006.

[3] D. Box. *Essential COM.* Addison-Wesley, Reading, MA, 1997.

[4] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, May 2000.

[5] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 31–39, New York, NY, USA, 2005. ACM.

[6] G.Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*. North-Holland Publishing Co., 1974.

[7] A. Goderis, C. Brooks, I. Altintas, E. A. Lee, and C. Goble. Heterogeneous composition of models of computation. Technical Report UCB/EECS-2007-139, EECS Department, University of California, Berkeley, Nov 2007.

[8] M. Henning and M. Spruiell. *Distributed Programming with Ice.* 3.3.1b edition, July 2009.

[9] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, Jan 2003.

[10] T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed Giotto. *SIGPLAN Not.*, 40(7):21–30, 2005.

[11] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. *Information Processing*, 1977.

[12] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, Oct. 2001.

[13] H. Kopetz and G. Grunsteidl. Ttp - a time-triggered protocol for fault-tolerant real-time systems. In *Proceedings of The Twenty-Third International Symposium on Fault-Tolderant Computing*, volume FTCS-23, 1993.

[14] B. Lewis and D. J. Berg. *Multithreaded Programming With PThreads.* Prentice Hall PTR, 1997.

[15] R. Monson-Haefel. *Enterprise JavaBeans.* O'Reilly, 3rd edition, 2001.

[16] Object Modeling Group. *Data Distribution Service for Real-Time Systems, Version 1.2*, formal/07-01-01 edition, January 2007.

[17] J. Prosise. *Programming Microsoft .NET.* Microsoft Press, June 15 2002.

[18] P. Puschner and C. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[19] I. Pyarali, C. O'Ryan, D. Schmidt, N. Wang, A. Gokhale, and V. Kachroo. Using principle patterns to optimize real-time ORBs. *Concurrency, IEEE*, 8(1):16–25, Jan-Mar 2000.

[20] I. Pyarali, D. Schmidt, and R. Cytron. Techniques for enhancing real-time CORBA quality of service. *Proceedings of the IEEE*, 91(7):1070–1085, July 2003.

[21] R. E. Schantz, J. P. Loyall, D. C. Schmidt, C. Rodrigues, Y. Kirishnamurthy, and I. Pyarali. Flexible and adaptive QoS control for distributed real-time and embedded middleware. In *Proceedings of Middleware 2003*, Rio de Janeiro, Brazil, June 16-20 2003. 4th IFIP/ACM/USENIX International Conference on Distrubted Systems Platforms.

[22] D. Schmidt, D. Levine, and S. Mungee. The design and performance of real-time object request brokers. *Computer Communications*, April 1998.

[23] A. Schuster and J. Sprinkle. Synthesizing executable simulations from structural models of component-based systems. In *3rd International Workshop on Multi-Paradigm Modeling*, October 2009.

[24] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the java system. *USENIX Computing Systems, MIT Press*, 9(4), Nov/Dec 1996.

[25] K. Wong and C. Wang. Push-Pull Messaging: a high-performance communication mechanism for commodity SMP clusters. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 12–19, 1999.

[26] B. Zeigler, H. Praehofer, and T. Kim. *Theory of modeling and simulation.* Academic Press, 2000.