THE SPECIFICATION AND IMPLEMENTATION OF A MODEL OF COMPUTATION

By

Ryan Thibodeaux

Thesis
Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

May, 2008

Nashville, Tennessee

Approved:

Professor Gábor Karsai

Professor János Sztipanovits

To my family,
for your unwavering love and support
through all my endeavors.

ACKNOWLEDGEMENTS

This work was sponsored by the Air Force Office of Scientific Research, USAF, under grant/contract number FA9550-06-0312 and the NSF/ITR, titled Foundations of Hybrid and Embedded Software Systems, award number CCR-0225610.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CNI	Communication Network Interface
$DEVS$	Discrete Event System Specification
ET	Event-Triggered
MoC	Model of Computation
TMR	Triple-Modular Redundancy
$TTMoC$	Time-Triggered Model of Computation
TT	Time-Triggered
TTA	Time-Triggered Architecture
TTP	Time-Triggered Protocol
VM	Virtual Machine

CHAPTER I

INTRODUCTION

The integration of computing devices into more and more of our surrounding technologies has been ongoing over the past two decades. The continuing trend in technology-driven fields (e.g. automotive industry, consumer electronics, automation, etc.) is to increase the available computational resources and software-based implementation in order to add functionality and lower costs. Unlike typical software systems targeted for personal computers and web-based applications, these *embedded systems* are subjected to physical constraints as a result of their necessary interaction with the physical environment and/or the limited availability of computational, communication, and physical (e.g. power) resources. This interplay between the continuous-time dynamics of the physical world and the discrete-time computational model of software only complicates the design process for embedded systems. Consequently, embedded software systems designers have tried to construct reasoning/modeling frameworks that leverage techniques from both continuous-time and discrete-time systems in order to alleviate some of the design challenges. Ultimately, any approach intended to design, model, and analyze embedded software systems can become an accepted technology only if it is adaptable to a variety of application domains and formal enough to provide logical assurances.

Typically in the design of software systems, partitioning a functional software specification into distinct modules or components is often considered a requirement of "good" software engineering practices. Each component provides some functional or behavioral utility, and the composition of the components across well-defined connections/interfaces is intended to result in the desired system behavior. Deriving any notion of the overall behavior from component-level descriptions is contingent upon an established understanding of the rules that govern the how the components are connected, i.e. the structure of the system and the legal interaction patterns between components that occur over time (the operational semantics). Generally, a collection of these rules targeted for a set of systems that share a common syntax is called a Model of Computation (MoC) [7].

Considering the MoC attributed to a design is fundamental to understanding the behaviors that can and will emerge during system execution, providing an unambiguous specification of the MoC and its influence over components is pivotal to successfully modeling run-time behaviors. Various tools and approaches have been developed to tackle the issue of describing MoC-s, each of which offers its own strengths and weaknesses depending on its underlying objective: mathematical rigor, simulation, verification, etc. We contend that all of these objectives are commendable and important to understanding embedded software systems; however, we hold that it should be primary to provide designers a mathematically precise reasoning framework that considers the operational description of a system fundamental to effective software engineering. Furthermore, effectively communicating the consequences of a component-based implementation over a chosen MoC requires that a modeling framework must be able to express characteristics inherently present in embedded systems, mainly heterogeneous classes of behaviors and temporally-aware computing.

Given the wide range of applications and complexity of embedded systems, it should come as no surprise that one computational model cannot be universally applied to describe all system behaviors and events. Instead, designers rely on a multitude of MoC-s for capturing different types of components and interactions. In the most general case, how and when components logically execute and communicate with each other can be described by two types of responses: event-triggered (ET) and time-triggered (TT). ET responses are reactive: the occurrence of a discrete event (e.g. the arrival of a message, an interrupt request for service, or changing a variable) elicits a response from the system as fast as possible or with some measurable quality of service. Conversely, TT systems initiate an event or activity only at predefined moments of time laid out according to a fixed schedule. The synchronization of TT events across components implies they maintain an agreement on the current state and ongoing progression of "system" time. ET systems are much more unpredictable, whereas TT systems are fundamentally deterministic. Unfortunately, most implementations of embedded systems cannot be classified as strictly ET or TT, but contain a mix of both depending on the function of a component or set of components (a subsystem). Obviously, the applicability of an approach for modeling embedded software hangs on its ability to sufficiently capture at least these general classes of behaviors (separately and intermixing them), and it must provide a mechanism to account for the continuous progression of time beyond discrete incremental changes of a fixed-resolution counter.

The need for capturing properties and characteristics of a MoC that greatly affect the run-time execution of a system has underscored the stated desirables of a modeling framework thus far. Attention to these implementation-specific details is often captured in a system model commonly referred to as an execution framework or *platform*. A platform provides the operational services dictated by the chosen MoC that orchestrate the behaviors and interactions of the constituent software components during execution. In embedded software systems, platforms can be real-time operating systems, middleware frameworks, component execution frameworks, or kernels. Many abstract specifications of software systems and MoC-s try to avoid if not outright reject the need to model implementation-specific details (e.g. protocols, timing, component scheduling, and synchronization mechanisms); however, there are specific embedded software domains where the precise knowledge and understanding of a platform and its execution logic is paramount to producing the desired system with a level of certainty regarding its performance and reliability. One such domain of systems is high-confidence real-time systems, where suboptimal performance or failures can result in loss of human life or property (e.g. flight control and avionics systems, automotive applications, robotics, manufacturing, health and safety monitoring, etc.).

Thesis Objective

Considering the behavioral and temporal challenges prevalent throughout embedded software system design, we propose the development of a modeling framework that can describe various MoC-s and their influences on component-based software systems with a focus on their operational behaviors that unfold over the progression of time. We require the use of a modeling language that is not only mathematically precise and conducive to reasoning techniques but also fully capable of expressing a system's state evolution as a result of the occurrence of events and the ongoing progression of time. Our proposed selection for the modeling approach, DEVS [10, 11], further extends our framework with the capability to rapidly prototype and simulate a modeled system (components, connections, and MoC(-s)) such that designers can evaluate resulting execution traces for logical and temporal correctness.

Furthermore, we will illustrate how to use the proposed modeling framework to develop an implementation of a strictly TT platform intended for deploying distributed real-time systems. A representative model of the platform will be constructed and then used to derive the execution logic of the platform, which will be implemented on off-the-shelf software and hardware components. We envision this exercise will show that modeling an execution platform intended for implementation has many benefits and consequences: it eases the process of developing the implementation by equipping the designer with a design blueprint, it allows evaluating the correctness of the implementation against the reference behaviors of the model, it provides system designers a framework to evaluate how MoC-s determine run-time behaviors of an application, and benchmarks/properties of the implementation can be integrated into the platform model to improve the fidelity of timed simulations.

CHAPTER II

BACKGROUND: DEVS MODELING FORMALISM

The Discrete Event System Specification (DEVS) formalism [10, 11] is a mathematical language intended to unambiguously describe time-driven systems. A DEVS system is characterized by states, input/output events, a time base, and functions that describe the evolution of the system state based on the occurrence of events and the passage of time. Like other discrete event formalisms, DEVS captures the changing variable values (event occurrences) over well-defined time segments; however, unlike most discrete event simulators, DEVS allows the instantaneous occurrence of events, i.e. the length of the time segments between events is variable (over continuous time) instead of over fixed time steps [12]. Initially, simpler (atomic) models are created to describe the fundamental dynamic behaviors of a system, and then larger (coupled) models are constructed from a network of the simpler models to produce a complete system specification.

Atomic DEVS Models

An *Atomic* DEVS model is represented by the 7-tuple structure [2]

$$M_A = \langle X, Y, S, s_0, \tau, \delta_x, \delta_y \rangle, \text{ where}$$

- X is a set of input events.
- Y is a set of output events.
- S is a finite set of discrete states.
- s_0 is the initial state.
- $\tau : S \to \mathbb{R}^+_{0,\infty}$ is the time advance function. It returns a non-negative real number (the lifespan of the current state) that indicates how long the system can remain in the current state without the arrival of new input events.
- $\delta_x : Q \times X \to S$ is the input transition function where

$$Q = \{(s, t_s, t_e) | s \in S, t_s = \tau(s), t_e = [0, t_s]\}$$

and t_e is the elapsed time since the last entering of state s.

- $\delta_y : S \rightarrow Y \times S$ is the output transition function. It specifies the next internal state of the system and the generated output events whenever the lifespan of the current state expires.

Time is a continuous variable in all DEVS models with a constant rate of one. Note, this does not imply that the lifespan of a state, $\tau(s)$, is also a continuous variable; instead, the lifespan is given only as a result of evaluating δ_x or δ_y, i.e. upon entering a state.

Two types of transitions describe the evolution of state for a given atomic DEVS model: internal and external transitions. *Internal* transitions occur when the lifespan of a state is reached and result in entering some new state and the generation of some set of output events given by δ_y. *External* transitions result from the arrival of a new input event that triggers a state change under the enabling conditions given by δ_x. External transitions occur instantaneously and generate no output events.

Coupled DEVS Models

Like other automata-based modeling languages, constructing complex systems using one atomic or flat DEVS model is possible but very tedious and cumbersome. This approach is not only laborious but also prone to error, lacks generality, and most likely results in an incomprehensible system specification. DEVS helps steer designers away from this path by promoting the use of hierarchical and modular model specifications within its generalized framework. A large DEVS model can be constructed by coupling or composing simpler DEVS models across well-defined interfaces. Through this coupling, events generated from one subsystem model can be passed “horizontally” (between peer systems) and “vertically” through the hierarchy to other subsystem models (where subsystems are atomic or coupled models as well).

A *Coupled* DEVS model is given by the 7-tuple structure [2]

$$M_C = \langle X, Y, D, \{M_i\}, EIC, ITC, EOC \rangle, \text{ where}$$

- X is a set of input events.
- Y is a set of output events.
- D is a set of names of sub-components.
- $\{M_i\}$ is a set of DEVS models where $i \in D$. M_i can be an atomic or coupled DEVS model.
- $EIC \subseteq X \times \bigcup\limits_{i \in D} X_i$ is a set of external input couplings where X_i is the set of input events of M_i.
- $ITC \subseteq \bigcup\limits_{i \in D} Y_i \times \bigcup\limits_{i \in D} X_i$ is a set of internal couplings where Y_i is the set of output events of M_i.
- $EOC \subseteq \bigcup\limits_{i \in D} Y_i \times Y$ is a set of external output couplings.

The semantics for the progression of time and state evolution within a coupled DEVS model follow directly from the semantics of an atomic DEVS model. Whenever M_C receives an input event, the coupled DEVS transmits the input event to the sub-components through the set of external input couplings, and when a sub-component produces an output event, the coupled DEVS transmits the output event to the other sub-components through the set of internal couplings and it also produces an output event of M_C through the set of external output couplings.

Summary

DEVS provides a general modeling formalism for describing discrete-event systems with semantic constructs for physical time, discrete events, discrete states, and hierarchical composition of models. Accordingly, its wide-scale applicability, mathematical underpinnings, and model simulation support provided by various tools and libraries (e.g. DEVS++ [2], DEVSJAVA [12], Adevs [8], etc.) make it a viable option as a modeling framework for capturing and investigating MoC-s and their effects on software systems.

CHAPTER III

DEVS MODEL OF A TIME-TRIGGERED PLATFORM

This chapter introduces the use of the DEVS modeling formalism for specifying MoC-s for execution platforms in an implementation-independent manner. Given a set of software components and their connections, a selected execution platform and its MoC determine the execution semantics of the individual components and the legal interactions between components. We believe that the following approach enhances a software developers reasoning about a system design by providing simulation/prototyping capabilities for models of software components and their couplings over a selected MoC. In this example the interested platform is a TT MoC motivated by the Time-Triggered Architecture (TTA) [4] of the Vienna University of Technology and Dr. Hermann Kopetz.

Time-Triggered Model of Computation

Once a chosen MoC is attributed to a software system, the MoC defines the structural, behavioral, and temporal properties that will characterize the system's behavior throughout execution [7]. Furthermore, the precise specification of a MoC is fundamental to understanding how system-level properties will arise from the component-level behaviors and interactions, i.e. compositionality. The use of the DEVS modeling framework is intended to leverage its precise mathematical foundations for unambiguously specifying MoC-s, and the operational nature of DEVS models will facilitate the analysis of modeled system behaviors and serve as a logical framework for migrating from models of MoC-s to their implementation.

The application domain of distributed real-time systems (industrial processes, aviation, automotive, etc.) depends on software-based implementations of physical control systems [3]. These applications are typically modeled by the following behavior: sense environmental/physical data (speed, heading, temperature, etc.), perform some process calculations based on the sensed input, apply an adjustment to the environment through actuators (motor, piston, pump, etc.), and repeat this process continuously. This continual operation of trying to achieve a desired reference level of performance through repetitive processing steps motivates the use of strictly periodic models of system behavior. Safety and reliability are of the utmost criticality for any implementation intended to meet the desired controlled operation for such systems; failures may result in physical harm (if not death) to humans, animals, or property. Consequentially, these expectations on such systems have motivated the use of techniques that will guide designers towards implementations that are analyzable and verifiable regarding system performance.

One such approach for building dependable, periodic control systems is through the use of the Time-Triggered Model of Computation (TT MoC) [3]. In a TT system, triggering signals are generated over the progression of time according to statically defined schedules produced in the design phase. The requisite *a priori* knowledge of when events and actions are to occur through the use of static schedules produces systems that are more conducive to predictive modeling and analysis. Typically, these systems are constructed by partitioning the software-based functionality into concurrently executing software components, called *tasks*, that execute over a set of computational hardware units or processors, called *nodes*, and nodes are connected via a network for passing data between tasks within *messages*. An execution platform that implements the TT MoC on a node will use the TT control signals to initiate/terminate task execution, start the transmission of data messages, and initiate interactions with the physical environment or other systems.

The TTA [4] developed at the Vienna University of Technology by Dr. Hermann Kopetz is an industrial-level development framework produced by TTTech Computertechnik AG that provides the aforementioned services of the TT MoC for the dependable execution of safety-critical real-time systems. TTA further maintains the dependable execution of TT systems by offering redundant communication buses, a fault-tolerant time synchronization algorithm, and infrastructure for the replication of hardware and software subcomponents. The TTA requires the strict synchronization of system time and real-time data (i.e. state of variables) across all constituent nodes along with the complete separation of the TT execution of software tasks from the TT communication protocol via a well-defined hardware interface (the Communication Network Interface or CNI). This separation along with the well-defined interaction mechanisms that determine how and when information may cross (if at all) the CNI are intended to further ensure the temporal and operational accuracy of executing systems. Furthermore, these restrictions support the use of the TTA as a composable architecture capable of handling the dynamic integration of nodes during run-time along with the use of heterogeneous node-level implementations in both hardware and software.

The logical and physical separation at the CNI allows a system built using the TTA to rely on various communication protocols for transporting messages between nodes. However, any viable candidate must be able to reliably transmit all TT messages according to the strict periodic message schedule with the necessary guarantee that data will be transmitted from the source to the receiver in the well-defined time interval between which the application-level tasks update the CNI and retrieve data from the CNI on their respective nodes. The TTA offers two Time-Triggered Protocols (TTP-s) as possible candidates: TTP/C and TTP/A. TTP/C [5] is intended for high-confidence systems that require the highly dependable transmission of TT messages between nodes and multiple mechanisms for avoiding faults or deleterious behaviors. TTP/A [6] is a low overhead implementation of a TTP that relies on a non-fault-tolerant master/slave configuration of nodes for passing TT messages. TTP/A is intended for lightweight sensor/transducer-level applications of distributed real-time systems, e.g. process control systems and low-level sensing in automotive systems.

In purely TT systems, the schedules that describe the tractable execution across all nodes are strictly periodic. For both task execution and message passing, the execution is separated into continuously repeating periods or cycles, called *hyperperiods*. Consequentially, this periodicity requires that the schedules specify only those timed events and actions that are invoked over a single hyperperiod. Each schedule sequentially orders all task execution or message instances in a given hyperperiod, and it is the foremost responsibility of the TT execution platform to maintain the timely execution of the schedules over each hyperperiod cycle while the system is operational. Due to the causal dependencies between tasks executing and their resulting data that gets passed in the scheduled messages, the hyperperiod must be the same for both schedules.

The TT MoC DEVS specification presented below captures the general TT execution services required for the strictly periodic execution of software-based control systems: scheduled execution of software tasks or components, scheduled transmission of data messages, and synchronization mechanisms for ensuring lock-step execution of the system across all nodes. As this specification grows, mechanisms for fault-tolerance and improved robustness common to platforms such as TTA will be included. However, the objective of this framework is to be flexible enough to investigate the heterogeneous composition of MoC-s that allow behaviors other than those offered by strictly TT systems; therefore, the abstract specification of the individual DEVS components is of critical importance.

Time-Triggered Platform Modeling Framework

We now present the corresponding DEVS model of an execution platform that provides the aforementioned services of the TT MoC for a networked system of nodes (see Figure 1). The following sections will describe the primary components running on a single node that are responsible for maintaining the scheduled transmission of messages and execution of software tasks, the TT communication controller and TT task scheduler respectively. Also, models of a communication bus and software tasks are provided in order to illustrate how the modeling framework could be used to prototype a specified system. In the current manifestation of the modeling framework, all of the DEVS models are constructed using the DEVS++ C++ library [2] created and maintained by Dr. Moon Ho Hwang.

Time-Triggered Communication Controller

The primary responsibility of a TT communication controller is to accurately follow a fixed message schedule. A message schedule is generated offline, passed as an input to the platform, and cyclically evaluated throughout the execution. Unlike the strict time division multiple access scheduling rules of TTA and the TTP/C protocol [4, 5],

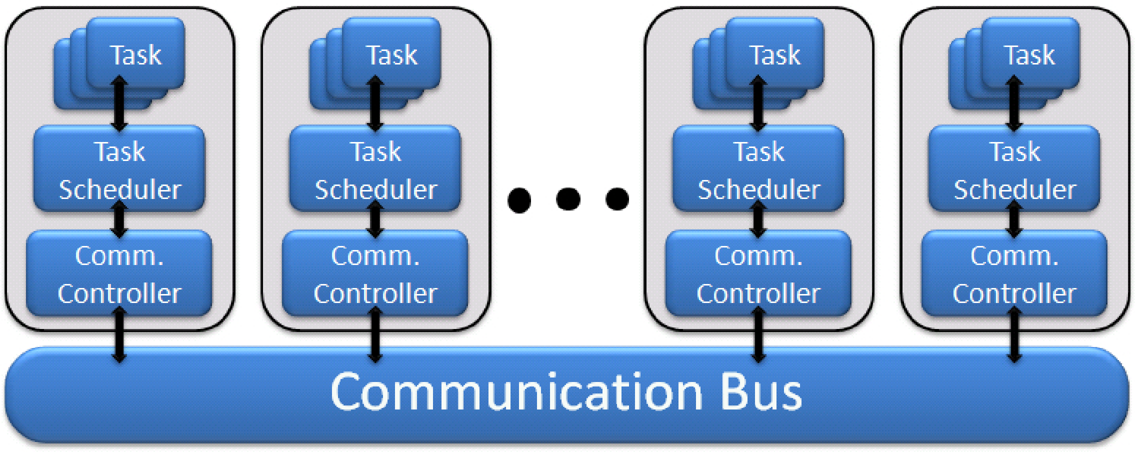


Figure 1: High-level architectural view of a TT system

this platform does not place any minimum or maximum on the amount of data that can be sent from a node in a round/cycle of the hyperperiod except those constraints that maintain the feasibility of the message schedule. Instead, the static message schedule specifies which node sends a message at an exact scheduled offset (the *phase*) with respect to the start of the hyperperiod. Each schedule round is one hyperperiod long, and the hyperperiod is of fixed duration. A node will be allotted no messages/slots if it is not expected to transmit any data within a schedule round. If a message is expected to be sent with a frequency greater than one within a round, it will be listed as many times in the schedule as it is expected with each appropriate scheduled transmission time. The length of each message, in bytes, is also indicated in the message schedule.

Establishing and maintaining TT communication across a network of nodes implemented using this platform proceeds according to the listed steps (see below). Figure 2 below is a DEVS model specification of the behavior as well. Within the DEVS representation, the italicized text (e.g. $f(msg.size)$) directly below a state label (e.g. **SYNC**, **RECV**, and **IDLE**) indicates the lifespan of its corresponding state. Also, a lifespan labeled as $f()$ is a function of its argument, and a lifespan labeled with φ represents a constant. Variables *next* and *HP_Time* are used to indicate the index (starting from 1) of the next scheduled message and the start time of the current hyperperiod respectively. The function *Time*() returns the current global time value. The event-triggered (external) and time-triggered (internal) transitions between states are represented by solid and dashed black arrows respectively. The semicircles on the boundaries of the model represent the event ports of the model: protruding ports are output ports and inlayed ports are input ports. Lastly, not present in Figure 2 but used in Figure 3, the subscript of a label (e.g. rel_i and $stop_{next}$) corresponds to an index value.

The logical execution steps for the TT communication controller proceed in following manner:

1. Synchronize all of the nodes on the network. Extremely important for ensuring the nodes begin executing the message schedule simultaneously. The amount of time to synchronize is dependent upon the underlying hardware and the chosen synchronization algorithm.
2. Transmit the hyperperiod start signal (marks the beginning of schedule execution).
3. Transmit data messages according to their strictly scheduled order and phase. Concurrently, allow reception of messages from other nodes. If a message is transmitted, ensure that a collision on the communication bus does not take place and proceed to the next scheduled event (either sending the next message or waiting for the end of the hyperperiod). If a message is received, process the message so that it can be used by software applications. The processing time depends on the size of the message data.
4. End of hyperperiod is reached (a schedule cycle has been completed).
5. Repeat process starting from step 2.

The requisite model (for simulating multi-node systems) of the communication bus is straightforward (see DEVS model in Figure 3). The model makes no assumptions regarding a specific hardware or protocol implementation for the bus, but all of the system nodes are expected to be connected to the bus for broadcast communication. Also, only one message can be on the bus at any given time; however, the message schedule for the TT MoC is typically constructed with this constraint in mind.

Indicated in Figure 3, the bus is initially idle and will remain so until a message is received on any of its *recv*() ports. In this model a unique triple of ports (*recv*(), *send*(),

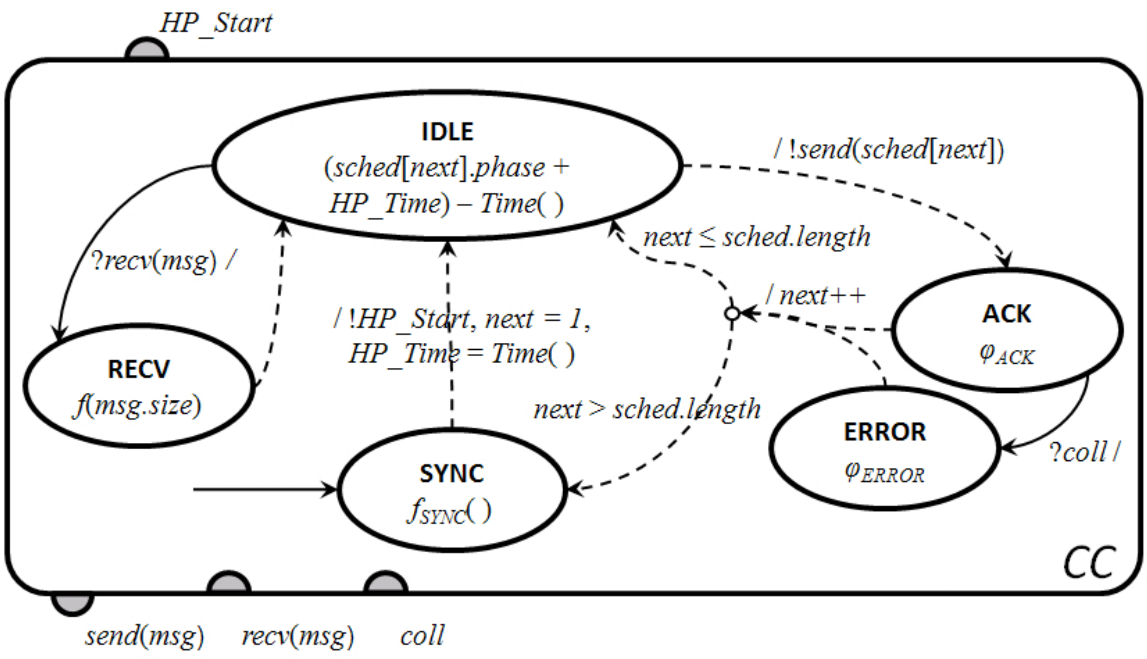


Figure 2: DEVS model of the time-triggered communication controller

and $coll()$) is created for each node in the system, and the nodes and their respective ports are differentiated by an index value (i and j in this example). Following the reception of a message, the bus will remain busy for some amount of time that is a function of the size of the data message, i.e. the transmission time over the bus. If no other message arrives during this time, the message will be sent to all nodes on the network. Conversely, if a new message does arrive from any node while the bus is busy, a collision occurs, the collision signal is sent to all nodes, and the data messages are lost.

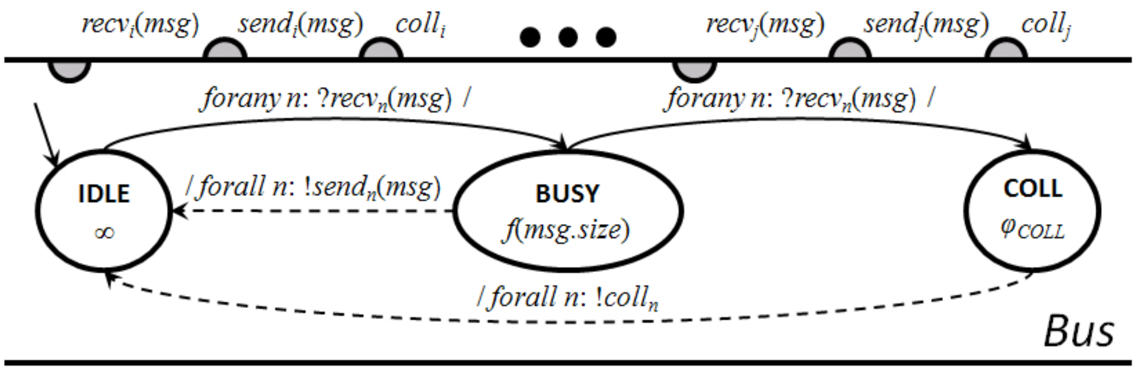


Figure 3: DEVS model of the communication bus

Time-Triggered Task Scheduler

Much like the communication controller, the TT task scheduler is primarily responsible for initiating the periodic execution of the software tasks based on a static task execution schedule initially passed as an input to the system. The current implementation model does not allow the preemption of tasks, i.e. only one task is released for execution at any given time and it finishes its execution before another task is allowed to begin executing. In order to properly maintain the timely execution of all tasks according to the schedule, the TT task scheduler is also responsible for terminating executing tasks that have yet to finish prior to reaching their worst-case execution time (WCET).

The task schedule for a node is similar to the message schedule mentioned in the preceding section. The schedule first specifies the hyperperiod that must match that of the message schedule. Next is an ordered listing of all task invocations. An invocation is listed for each time a task is to execute within a hyperperiod, and each listing specifies the scheduled release time (the phase) and WCET of the task invocation. Synchronizing the execution of the task schedule with the message schedule maintains the correctness of execution with respect to the system performance; therefore, each new round of the task schedule has to be initiated by the arrival of an event from the underlying communication controller indicating that a new schedule round is beginning. If no such event is received, then no tasks will be released for execution.

Not surprisingly, the implementation logic of the TT task scheduler is not dissimilar from the TT communication scheduler. Following along with the DEVS model in Figure 4, the execution steps are:

1. Synchronize the execution of the task schedule with the message schedule. Task execution cannot begin until the hyperperiod start signal is received from the communication controller.
2. Release tasks for execution according to their scheduled order and phase.

3. Upon the halted execution of the previously released task, move on to the next scheduled event: either the next task invocation or resynchronizing with the communication controller. Halted execution of a task is the result of either its self-completion or forced when its WCET is reached.
4. End of the hyperperiod is reached.
5. Repeat process starting from step 1.

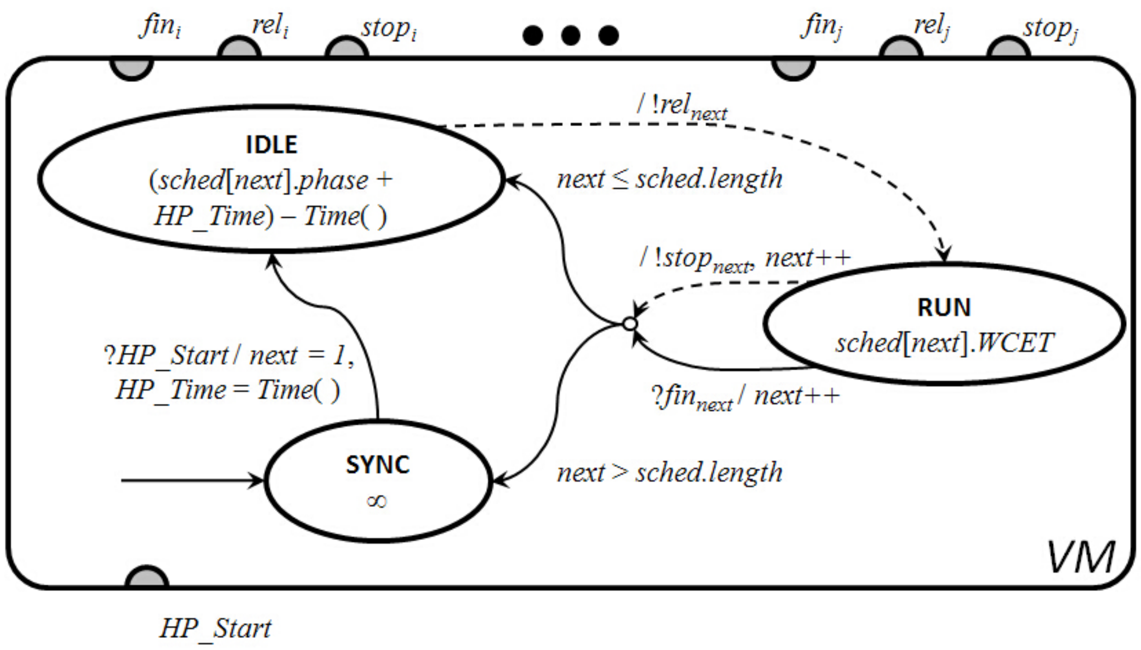


Figure 4: DEVS model of the time-triggered task scheduler

The DEVS TT task model (see Figure 5) is very simple since it is not intended to capture the actual logic of the software or how data is manipulated; instead, it merely supplies the timed execution traces for TT tasks. As indicated by the model, a task will remain idle until it is released for execution by the task scheduler. The task will continue executing until it finishes or it is forced to terminate by the TT task scheduler. The execution time of the task (the lifespan of the state **RUN**) is modeled as a probabilistic function based on the task's WCET. On an actual system, the execution time of a software task would be dependent upon many factors (hardware, software, and environmental activity); however, the DEVS model must make the assumption that the WCET parameter was determined as a reasonable estimate based on the possible execution profiles of the system.

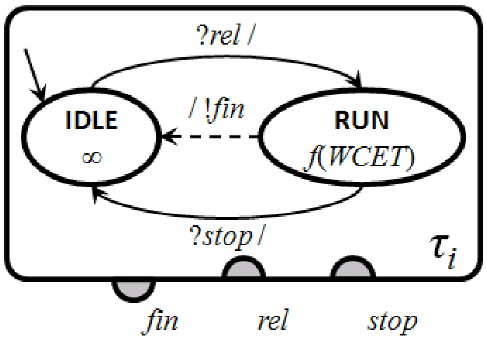


Figure 5: DEVS model of a time-triggered task

Putting all of these components together yields a coupled DEVS model for a single node that is presented in Figure 7. Also, Figure 6 is an example snapshot of a continuously executing TT system representative of the periodic execution of TT schedules. In Figure 6 each labeled time $t_{i,p}$ indicates the occurrence of event i in the hyperperiod round p and Δ_i is the events corresponding durative action time (e.g. transmission delay of a message or execution time of a task).

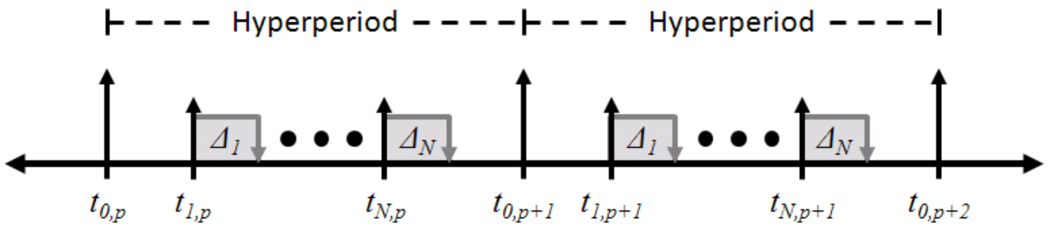


Figure 6: Activity timeline over the execution of a time-triggered schedule

Summary

In this chapter we have presented a modeling framework for describing MoC-s and execution platforms in a non-implementation specific context using the DEVS modeling formalism. This framework is intended to facilitate the specifying of MoC-s and their interactions in a mathematically precise and analyzable way, and the use of DEVS and its various tools allows software developers to construct executable models of their software systems. With this prototyping capability, developers can evaluate how a chosen MoC influences the behavior and temporal properties of their software applications.

Furthermore, an example MoC based on the TTA [4] was specified using the modeling framework. This platform, commonly used digital control systems, is based on the strictly periodic execution of software tasks and transmission of data messages to promote deterministic system performance. The underlying logic responsible for ensuring the behavioral characteristics of the MoC along with a template for modeling software tasks were provided to illustrate how to construct a model of a single node, and a simple communication bus model for sharing data between such nodes was provided for building larger models of networked nodes (a distributed system). The following chapter will present an implementation of the discussed TT MoC on off-the-shelf hardware and software components that used the preceding DEVS model as a logical guide for developing the execution platform.

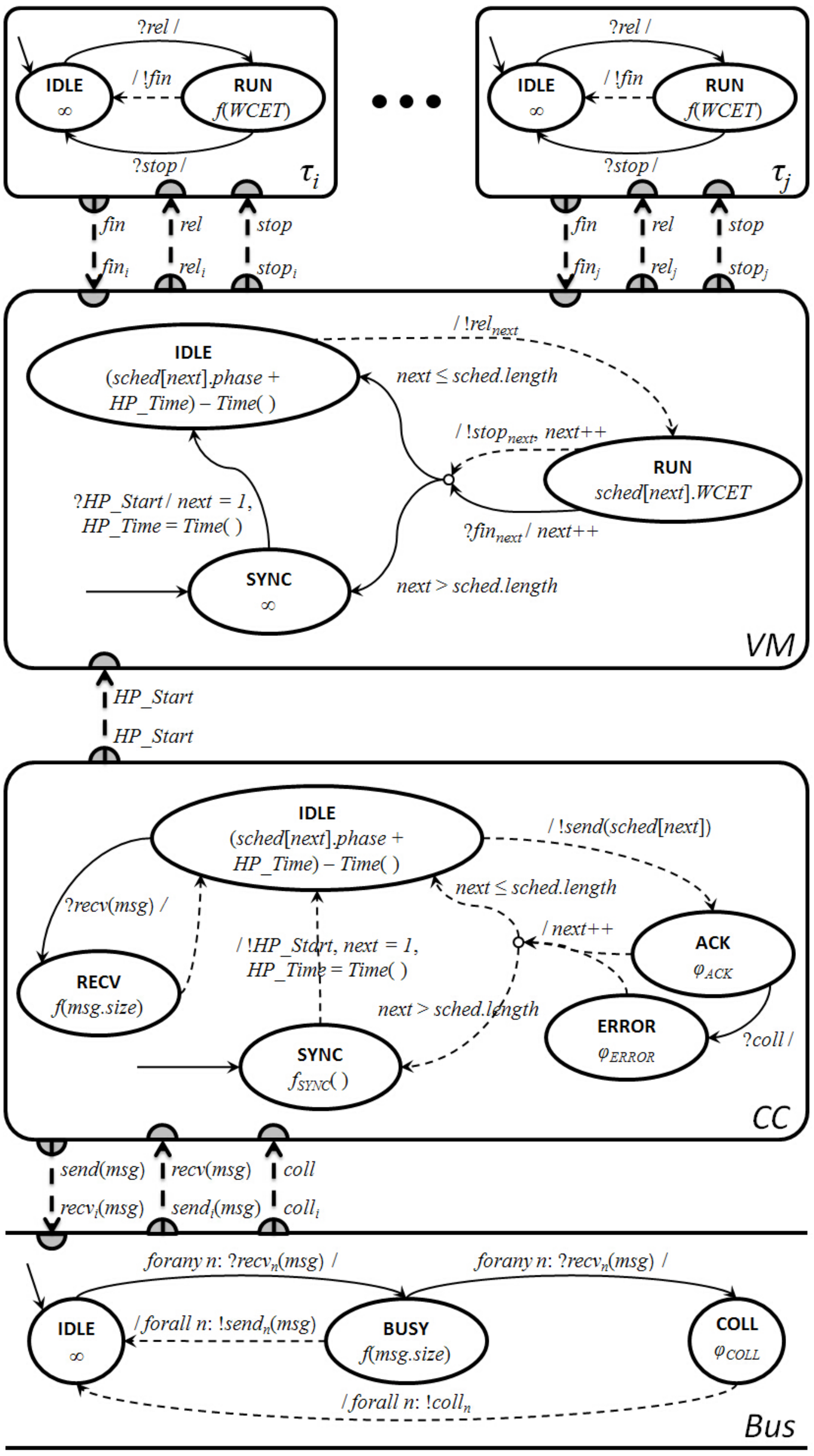


Figure 7: Coupled DEVS model of a TT MoC equipped node and the bus

CHAPTER IV

AN OFF-THE-SHELF PLATFORM IMPLEMENTATION

This chapter presents how the previously introduced TT MoC was implemented on off-the-shelf hardware and software components as a possible target execution platform for periodic digital control systems. Although the TTA has been developed and matured over many years in both research and industrial settings, it lacks the flexibility and extensibility for introducing other MoC-s or behaviors into the execution platform. These restrictions motivated the following execution platform that offered the same deterministic execution of the TTA with a hardware/software-independent platform Application Programming Interface (API). The resulting platform uses an abstract Virtual Machine (VM), FRODO, for executing TT software tasks coupled with a communication controller that provides the TT transmission of data messages between networked nodes, all of which is implemented using readily available infrastructure.

Platform Implementation Architecture

The platform's implementation architecture is based on the logical separation principles similar to those of other TT platforms like Giotto [1] and TTA [4]: the periodic execution of software tasks is determined solely by the passage of time and the current operational mode of the system, i.e. events such as the arrival of new data messages do not affect the timing of task execution. Accordingly, only the most recent data values, representative of some control signals, are relevant to calculations performed by TT tasks; therefore, a globally shared memory construct is used to read-in and update data throughout execution. The resulting architecture is provided below in Figure 8. In the figure, solid black arrows represent the flow of data throughout the system whereas the dashed black arrows (see TT Task Scheduler) indicate the sending of events.

As indicated at the lowest level of Figure 8, a shared bus architecture is used for the transmission of TT messages between nodes. There are no explicit restrictions placed on the communication protocol used; however, it needs to be a relatively high bandwidth connection that can be isolated from other networks to ensure communication integrity with as little overhead as possible. Currently, a standard Ethernet connection utilizing UDP broadcast communication is used to connect nodes in an isolated network across an Ethernet hub, a Netgear DS104 (with 4 10/100 ports). The physical nodes used in the deployment are Soekris net4801 units. These embedded boards come with a 266MHz 586 processor, 256MB of SDRAM, 3 10/100 Ethernet ports (each with an individual network interface controller), 2 serial ports, and 12 general purpose I/O pins. Also, each board is running a Linux 2.6.x kernel with no supplemental real-time patches. Linux was chosen since it is not only free but also widely accepted for embedded applications; however, in the near future, other deployments will be evaluated with an emphasis on porting the execution platform to other hardware/software configurations with an interest in operating systems that offer more “real-time” execution capabilities. Also, the selected programming language for the current implementation was C++.

Also illustrated in Figure 8 is the shared memory structure used to hold the relevant data that results from the calculations of the TT tasks (see the box in between the TT Communication Controller and TT Tasks) and is passed between nodes in the TT messages. This data structure contains one memory location for each unique message instance defined in the message schedule. Over the execution of a hyperperiod, all of the nodes should maintain the same data set, i.e. each node updates the respective memory location of a received data message regardless of which task (even if it is not on this node) will potentially use the data. The data is persistent as long as it is not updated; however, it is immediately overwritten whenever new data is available, i.e. there is no queuing of data. The representation of the shared memory in Figure 8 also indicates that the memory locations must be accessed controlled (see the white ”X” across the access point on the top of the box). This is to prevent race conditions if both a TT Task and the TT Communication Controller are trying to update the same memory location. Currently, a POSIX mutex variable that functions as a binary semaphore is used to regulate access to the shared memory. In future extensions to the platform, it would be advantageous to pursue other approaches to synchronizing/regulating access to the shared memory that did not involve potential blocking conditions on the calling threads. One approach could use two buffers and an indicator variable: the indicator points to the buffer that was previously updated by the last write operation and is currently ready to provide data upon a read operation. The next write operation updates the other buffer, and the indicator variable will switch to the newly updated buffer when the write is complete. This switching occurs with every new write operation. Accordingly, a write operation, even for large data sizes, never blocks a concurrent read operation, unlike the mutex.

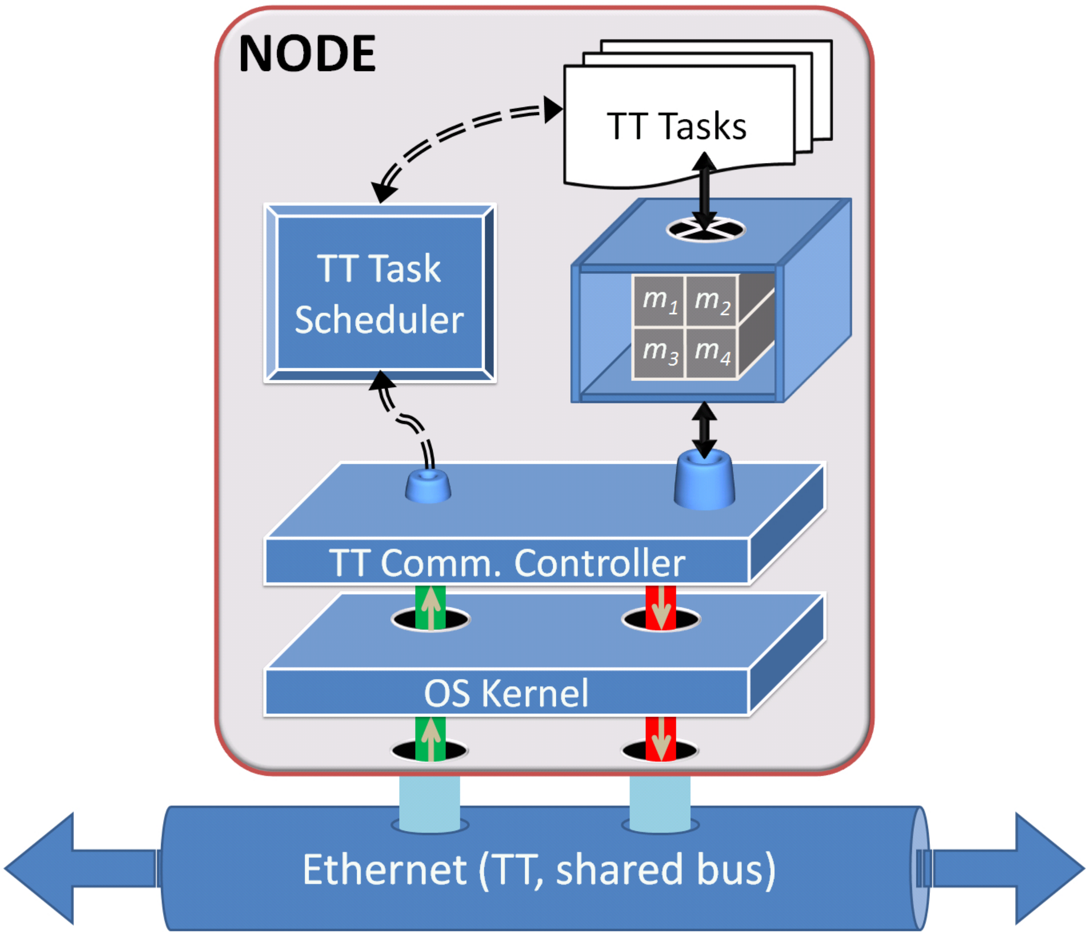


Figure 8: System architecture of a single node

From the details presented thus far concerning the implementation, we see that the active components running at any given time on the platform can be the TT communication controller, the TT task scheduler or VM, and any currently active TT task. Consequently, the implementation must provide mechanisms for handling the concurrent execution of these entities. Linux provides two types of such concurrency objects: processes and POSIX threads. POSIX threads were selected for this implementation since they require less overhead and a suite of threads in the same process share the same memory space. Furthermore, they do not place any restrictions on the use of C++ as the selected programming language.

Throughout the rest of this chapter and the following one, a common system configuration (see Figure 9) is used for experiments and benchmarking the execution platform. There are up to four independent nodes connected via a shared Ethernet network. Anywhere from zero to three nodes will have some sensing device used as a data input and either zero or one device will have an actuator output device. Each specific example will be briefly described when it is used, but this is the common configuration.

Concerning the experimental configurations as well, for reasons that will become evident in subsequent sections, an additional input to the system is required at run-time beyond the already mentioned message and task execution schedules. This additional input is a configuration file that lists the set of nodes that are expected to be on the network and part of the system during operation. The file merely contains unique identifiers that are used to differentiate between the nodes, and each node is provided its corresponding unique identifier during system initialization. The system will not go into operation unless all nodes are present and each node matches with a specified unique identifier.

High Resolution Timeouts and Scheduling in Linux

Linux supports halting the execution of a thread for an arbitrary amount of time with microsecond resolution using the clock_nanosleep() function. This function uses a predefined POSIX timer object as the reference clock for timeouts, and it can timeout for a specified amount of time or until it reaches some specific time value in the future. This function was used as the baseline method for scheduling events on the platform since the typical applications required a timing resolution around the one half to one millisecond range; however, this function was not without its limitations.

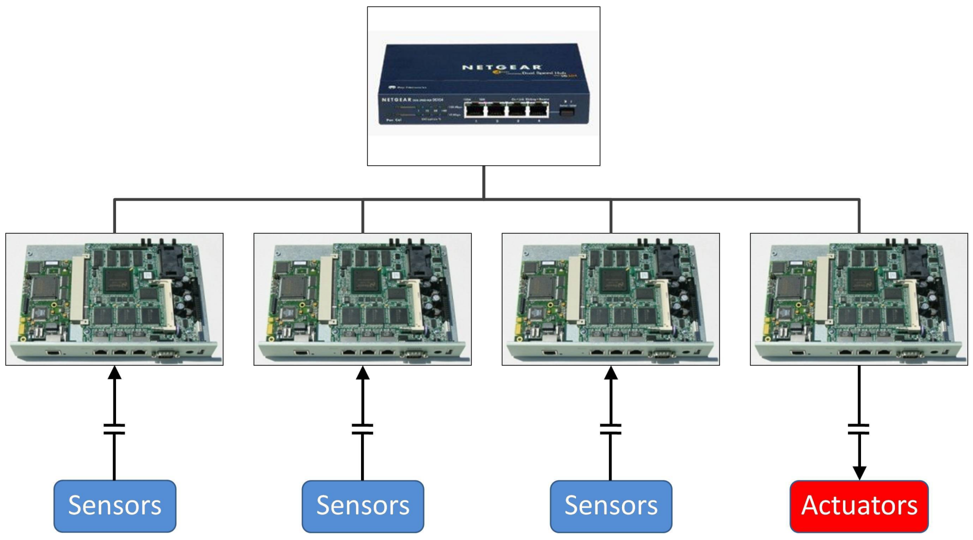


Figure 9: General implementation configuration of nodes for experiments

First, the Linux documentation indicates that even though the scheduling and timing mechanisms behind clock_nanosleep() guarantee that it will never return from a timeout prior to the specified target time, it provides no guarantees regarding how long after the target time it will return. This is a non-ideal implementation for real-time systems with strict timing requirements; however, we had no other available options using Linux routines or access to hardware-level interrupts and timers that would have provided a higher-level of timing safety.

Considering the general applications that would be running on the platform (mainly a suite of strictly periodic threads) and the isolation of the system, it became apparent that the scheduling overhead and computational demand on the system during execution would be periodic if not nearly constant. Further analysis of the system and use of the clock_nanosleep() function indicated that there is a baseline or minimum overshoot from the target timeout value, and this error typically stabilizes to a slightly larger value during system operation. The minimum overshoot or baseline resolution of the function was determined to be about 40μs. This implies that no matter what non-negative, non-zero value is specified as the target timeout value, the function will return no sooner than 40μs after it. Unfortunately, the error encountered during operation is typically computational load dependent; therefore, a dynamic adjustment approach was implemented to try to minimize this error to ensure the most accurate timing as possible for scheduling. The resulting high-resolution timeout routine was implemented using the following process.

First, the clock_nanosleep() minimum offset is determined during system initialization using test calls. All subsequent calls to timeout a thread (the communication controller or VM thread) are reduced by the current offset value for that thread (note: a different offset is maintained for each thread). If the desired timeout is less than the current offset, the system will busy-wait until the desired time is reached and then return. Otherwise, the offset reduction is taken and the new value is passed to clock_nanosleep() as the new desired timeout value. Upon returning from the timeout, the error is measured (difference between the current time and the desired time), and the offset value is adjusted by adding or subtracting some proportion of the error as long as the error is not too large such that the new adjustment would cause instability or oscillations in subsequent calls, i.e. extremely large errors must be ignored since the system will most likely not produce similar results in subsequent calls.

This offset adjustment approach is similar to using a proportional feedback controller from control theory to correct for operational errors; however, this implementation would have to be modified to fit a strict proportional controller model in order to discuss the stability or performance (e.g. steady-state error) of the controller. To be analyzable as such, the use of a discrete-time proportional controller would require that the timeouts would have to be strictly periodic with a fixed timeout value and continuously running even when a timeout by the communication controller or VM has not been requested. This option was explored as an interesting application of a control-theoretic approach to achieving accurate sub-millisecond timeouts in Linux; however, the overhead from running the adjusted timeouts with periods ranging from $150 - 500\mu$s resulted in starving the other executing threads of computation time and increased the response time of the communication controller and VM threads.

Since the system load is currently very predictable during execution, the adjustment value typically reaches a stable value very quickly once the execution of the schedules begins, i.e. there is little ramp-up time or "pipeline filling". Testing has shown that this method is accurate enough to yield consistent timeout errors less than 20μs. Unfortunately, as the system's load and external interactions with other networks and/or applications become less predictable, this correction scheme will no longer be effective or nearly as accurate. At such a time, the need for fine-grained timing control (hardware-level access and/or interrupts) will become a necessity to meet the strict timing demands of most TT systems.

Communication Controller

Given a system configuration and the cyclic message and tasks schedules, it is the responsibility of the TT communication controller on each node to maintain communication with the other nodes of the system and their partnered TT task schedulers in order to properly execute over the schedules in a synchronized manner. We will now detail the specific implementation approaches and performance benchmarks of the current off-the-shelf platform implementation using the previously detailed hardware/software configuration.

Initialization and Node Discovery

We now provide the implementation details of how the system nodes initially synchronize to confirm all are ready to begin executing. In this preliminary version of the execution platform, a fault-tolerant algorithm for establishing and maintaining node synchronization was not utilized since this is a non-trivial task in of itself for embedded and real-time systems. Instead, a simple master-slave approach that makes assumptions concerning the possible execution conditions and bus activity was chosen; however, improving the approach to be more robust (like the method used in TTA) would be worthwhile.

At initialization, each node must determine if it is the "master" node or one of the "slaves". Currently, the master is assigned to be whichever node has the first unique identifier listed in the system configuration file, and accordingly, all other nodes assume the role of a slave. As the designated master, a node must first ensure that all of the nodes listed in the configuration file are present before execution is allowed to begin (the node discovery process). The master node begins the discovery process by transmitting a synchronization message onto the network. The execution can proceed only if the master receives a synchronization acknowledgment message from each and every slave listed in the configuration file. Until all acknowledgements are received, the master will continually transmit the synchronization message every (N+1)*5 milliseconds, where N is the number of slave nodes. Conversely, when a slave node starts its initialization and reaches the node discovery step, it will wait indefinitely until it receives the synchronization message. Upon receipt, it will transmit its synchronization acknowledgement message at *pos**5 milliseconds after receipt, where *pos* is the integer index of the slave node's unique identifier in the configuration file. Accordingly, the discovery process is complete when all slaves have acknowledged the synchronization message, and the master node will conclude the process with the transmission of multiple completion messages.

Figure 10 shows an oscilloscope output of the message activity for this process with four nodes (note: the major intervals of the time axis are 10ms apart). The top channel corresponds to the master and the three other channels are the slaves. The brief pulses on each channel represent the transmission of a message. Initially, the master transmits the synchronization message, each slave then acknowledges the message at the proper 5ms intervals, and finally, the master sends out four synchronization complete messages at 20ms intervals (tries to ensure that all slaves receive this completion signal). This process is fairly rudimentary; however, all experiments on an isolated network have never failed to properly synchronize during initialization.

Synchronization of the Hyperperiods

Once the master is sure that all nodes are present, it is now responsible for initiating and maintaining a synchronized start of the hyperperiod (message schedule execution) for every cycle the system is operational. This is critical to maintaining functional correctness of the system, because the misaligned execution of the message schedule could increase the latency between the sampling of an input and the subsequent application

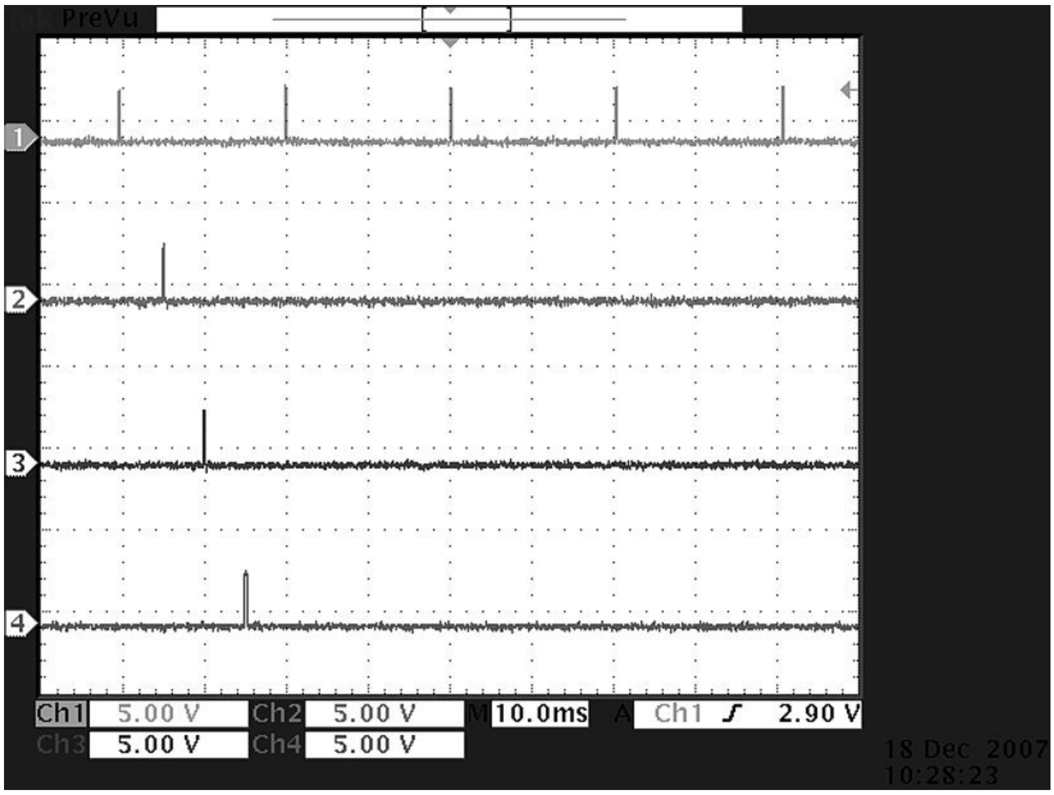


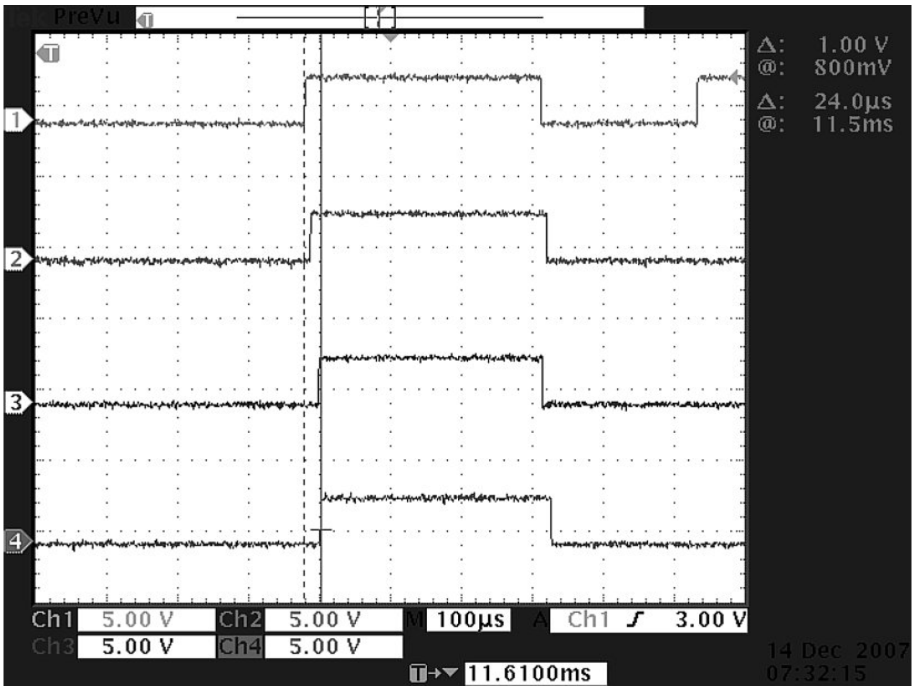
Figure 10: Node synchronization process viewed from the bus

of a control-related adjustment on the output of the system, i.e. degrade controller performance. Unfortunately, the current implementation's selected infrastructure (Linux, POSIX threads, non-hardware level timers, etc.) constrains the allowable overhead or computational requirements that the execution platform can introduce; therefore, the current method for synchronizing the hyperperiods is not fault-tolerant or robust. Ideally, the migration of this platform to more "real-time" capable hardware/software will enable the development of more thorough synchronization methods.

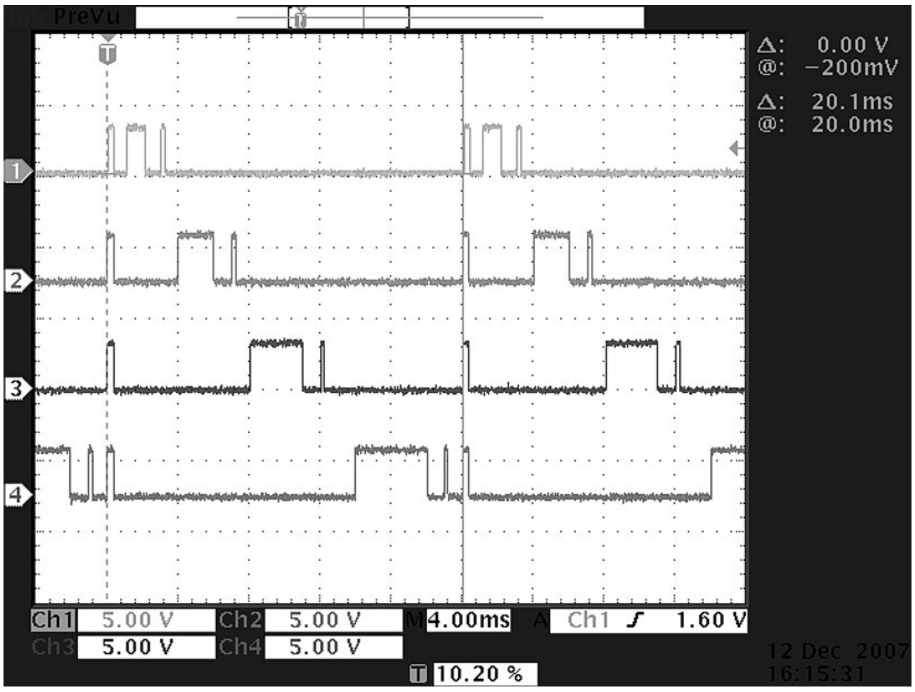
Similar to the discovery process, the slave nodes will not start the upcoming hyperperiod unless the appropriate message is received from the master node. After completing the discovery process or concluding a previous schedule round, the master transmits the hyperperiod start message on the network, and each individual node will immediately begin executing the message schedule after receiving this message, including the master. The master receives the message as well since the UDP broadcast protocol will send it to all nodes in the addressed subnet regardless of the sender. Once the message is received, each communication controller stores the current time as the start epoch of this round of the schedule and then transmits the scheduled TT messages that originate from its data set. The precision of this approach rests squarely on the uniform arrival time of the message to all of the nodes, including the master. This too is a naive approach to maintaining correct system performance; however, the current implementation characteristics (especially the restrictions on outside network activity) seem to provide a stable environment for yielding only marginal errors in the accuracy of the start time of each hyperperiod (see Figure 11(a)).

Figure 11(b) is another oscilloscope output of the activity of a set of four nodes over two hyperperiods of execution. Ignoring the details of how to interpret the various pulses on each channel, attention must be directed towards the two vertical cursor lines (a solid line on the right and a dashed line on the left). The cursors are positioned at the start of two hyperperiods, and the vertical pulse on each channel that is aligned with each cursor represents when each node began executing the hyperperiod. In this example the hyperperiod was 20ms and channel one corresponds to the master node. We can see at both hyperperiod synchronization instances that all four nodes begin executing at nearly the same time (usually within 30μs for all nodes, see Figure 11(a)). Also, the two cursors are placed to measure the time interval between these instances, and we can see that the measured time interval between the synchronization points was near 20.1ms. This error of about 100μs represents a deviation of 0.5% for this single instance; however, further data analysis over thousands of hyperperiods of varying lengths yielded an average hyperperiod error of about 40μs with a standard deviation on the order of 10μs. This means that it will typically take greater than 20,000 hyperperiods or schedule rounds before the accumulated deviation of the execution reaches one second.

Currently, this error between consecutive hyperperiods goes uncorrected in terms of when the messages are scheduled in the currently executing hyperperiod, e.g. the transmission of a message scheduled to occur $1ms$ after the start of the hyperperiod, t_{HP}, will occur at time $t_{HP} + 1ms$ even if the current hyperperiod started $50\mu s$ later than it should have (the corrected time would be $t_{HP}+1ms-50\mu s$). A more robust implementation (like TTA) with better clock synchronization and/or hardware-level timing capabilities could



(a) Difference in hyperperiod start times



(b) Length of hyperperiod

Figure 11: Hyperperiod synchronization across four nodes

account for this deviation (or possibly provide perfectly periodic execution); however, the inability to truly guarantee how accurately the master can send out the synchronization message (based on a timeout) or how quickly the communication controller can be notified of the arrival of a new message (from the select() routine) make adjusting for this deviation a risk with serious ramifications in tightly scheduled systems (e.g. skipping message instances).

Since the synchronization is based on the transmission of a message across the network, the delay introduced by transmitting a message has to be accounted for when the master node is determining what time to schedule the transmission of the synchronization message. Network isolation allows this delay to be compensated for using a fixed offset to indicate how long before the start of the next "ideal" hyperperiod that the synchronization message should go out on the bus. Through extensive testing and benchmarking, the current conservative compensation value was determined to be 250μs. This is considered conservative since it produces the presented synchronization accuracy with very few cases where the start of the next hyperperiod occurred earlier than the ideal time. Larger compensation values have been tried to reduce the average error to near zero; however, they always increased the likelihood of starting a schedule round early.

Synchronization of the Communication Controller with the VM

The simultaneous execution of both the message and task schedules is one of the primary requirements of the TT MoC for maintaining proper controller performance (keeps the tasks operating on the most current data). Upon establishing the start of a given hyperperiod across the nodes, the communication controller of each node must further synchronize the start of the hyperperiod with its respective TT task scheduler or VM (see "HP_Start" event in Figure 7). Given the selected Linux platform, the simplest and least computationally restrictive method of signaling to the VM to begin executing the task schedule was through the use of a POSIX mutex (a binary semaphore). The specifics of the VM execution semantics will follow in a later section, but we briefly mention some details here.

The communication controller is always given possession of the mutex during the system initialization process. Directly following the synchronization of the hyperperiods across all of the system nodes, the communication controller releases the mutex and briefly halts execution for a specified amount of time (currently 350μs). This pause allows the underlying operating system to reschedule the thread execution to allow the VM thread to acquire the mutex long enough to save the current time value as the start of its hyperperiod. The VM immediately releases the mutex, the operating system reschedules the awaiting communication controller thread, and the communication controller reacquires the mutex and begins executing over its message schedule. Each subsequent hyperperiod must be synchronized between the two threads as well; therefore, the VM must try to reacquire the mutex at the end of each its hyperperiods (it will be blocked) such that it is ready to save the new hyperperiod start time the next time the communication controller releases the mutex.

Obviously, using a mutex introduces a time lag between when the communication controller stores the current time value for the start of the hyperperiod versus when the VM can do the same (since the operating system has to reschedule the threads); however, this error can also be mitigated by subtracting a fixed offset value from the VM's saved hyperperiod start time. Once again, the predictability of this error results from the restriction that no other activity will be taking place on either the communication bus or the node itself while the system is operational (common underlying assumptions of the TT MoC for digital control applications). The current offset adjustment value is 130μs and the measured average error between the saved hyperperiod start times for the two threads is typically 15 – 20μs with very little deviation. This too is a conservative adjustment value, i.e. the message schedule runs slightly ahead of the task schedule; however, the TT MoC (especially in TTA [4]) dictates the primacy of accurate data sets over the scheduled execution of tasks. Once again, the values of 350μs and 130μs for this process had to be determined by benchmarking the chosen platform since both are highly software and hardware specific.

Transmitting and Receiving TT Messages

Now that all of the main tenants of synchronizing the communication controllers have been presented, the process of transmitting and receiving TT messages can be discussed. The logical operation of this process has already been presented in the DEVS description of the communication controller (see Figure 2); it is similar to most cycle executive operations over a pre-defined schedule. The communication controller properly transmits messages according to the sequential message schedule, and it must also listen for any incoming data messages on the communication bus while waiting to send out the messages. When a scheduled transmission time is reached, the communication controller obtains the current value of the data message from shared memory, packages the message for transmission, sends the message out on the shared communication bus via the underlying communication library, and moves on to repeat the process for the next scheduled message or awaits the start of the next hyperperiod if the end of the message schedule has been reached. Even though the high-resolution timeout mechanism (adjusted clock_nanosleep() routine) discussed above is capable of accurately scheduling the communication controller to awake when it is time to transmit a scheduled message, it cannot concurrently notify the communication controller when an incoming message has arrived via the communication bus. A different mechanism is required for this behavior, and it is provided by the select() function.

As a brief note, we would like to indicate how a message is packaged for transmission such that the reception side of message passing can be as quick as possible. The current implementation first retrieves the message data from the shared memory, and then the following string is attached to the front of an outgoing message: "(*msg_pos*)". Here, *msg_pos* indicates the index of the message in the message schedule. Since all nodes use the same message schedule, any node that receives this message can quickly determine from this index value which memory location to update with the new data. This implementation was also intended to add as little overhead as possible to the message communication process.

The select() function allows us to specify that we would like to listen to a communication endpoint (a UDP socket in this implementation) for any activity, i.e. the arrival of a new data message. If any message comes in on this endpoint after calling select(), it will return from being called indicating that a new message is available. Furthermore, a timeout alarm can also be specified that will cause select() to return if the timeout expires and no new message has yet been received (a timeout of length "null" will allow select() to wait for a new message indefinitely). This seems to provide all of the features needed by the communication controller to schedule message transmissions and concurrently listen for new data messages; however, the underlying implementation of select() is not without its limitations.

According to the Linux documentation, the timeout feature of select() is influenced by the operating system's timer interrupt. Regarding the select() function, this periodic signal, called a "jiffy", indicates the periodic points in time that the operating system will check to see if the timeout of the select() has expired. Our Linux kernel uses a jiffy with a period of 4ms; therefore, select() will only check the timeout alarm's status every 4ms. Obviously, any application that requires sub-millisecond resolution for scheduling cannot depend purely on this function, and such is the case for the TT communication controller. However, it does not mean the select() function's capabilities cannot be used if properly handled.

Instead of calling select() when a timeout must be specified for sending the next message and listening to the communication bus, the communication controller simply calls a "listen" routine and indicates the target time of the next scheduled event. This routine will use select() as described in the previous paragraph only if the timeout is greater than two times the jiffy period (8ms in this case). Any less of a timeout runs the risk of not being properly handled by the scheduler since a timeout that expires in between two jiffy signals is not handled until the later one. For example, a timeout of 4ms that starts 100μs after the jiffy signal will actually last 7.9ms. When it is safe to use select(), the new timeout value (TO_{new}) is an integer multiple of the jiffy period (P_{jiffy}) given by the following equation.

$$TO_{new} = P_{jiffy} * (\lfloor TO_{curr} \div P_{jiffy} \rfloor - 1) \tag{1}$$

When the desired timeout is less than 8ms, it is necessary for the listening routine to busy-wait and periodically check the communication endpoint for a new message. Busy-waits can be very expensive if they poll too quickly, but they can also increase response time to new messages if they do not check often enough. When the minimum allowable time interval between consecutive messages is 1ms (this is the typical constraint placed on all experiments thus far), a busy-wait period of 400μs was determined to be quick enough without using up too much processing time such that other threads (such as the VM) were starved.

After repeatedly performing the busy-wait operation, eventually the busy-wait period will be larger than the amount of time until the desired target time is reached. When this occurs, a timeout for the exact amount of remaining time is requested. Both the busy-wait calls and the final timeout request are handled using the offset-adjusted clock_nanosleep() routine described earlier.

Timing Properties of Message Passing

Throughout the description of the communication controller's implementation on the deployment platform of the Soekris net4801 boards running the Linux operating system, various detailed timing properties or limitations were provided to give a reasonable expectation of how accurately this off-the-shelf implementation could follow a predefined message schedule given certain assumptions. While these details are of the utmost importance to the schedule generation process for determining the schedulability of a designed system, the timing properties of physically sending, receiving, and processing message data on this implementation have yet to be discussed. Not only are these properties extremely important to schedule generation but they also exhibit the most variability.

Through various experiments and tests, the logical correctness of this implementation has been proven and its degree of accuracy in scheduling events has been quantified. Now, we must look at how the chosen platform affects the main phases of passing a message between nodes: from memory to the communication medium (transmission delay), across the communication medium from transmitter-to-receiver, and from the medium to the memory of the receiver (reception delay).

In order to gather results over a range of message sizes that could potentially be used in real-world applications, the following data was gathered assuming communication buffers of 4096 bytes (or 4KB). These are the buffers used by the underlying communication library (responsible for implementing the UDP protocol) that interacts directly with the operating system to move data on and off the communication medium. This buffer size is also used in the communication controller for retrieving message data from the shared memory; however, the actual size of the messages in memory is specifically provided in the message schedule and not equal to the buffer size. The only stipulation is that all messages must be of a size less than the message buffer, i.e. splitting-up message data across multiple messages is not allowed.

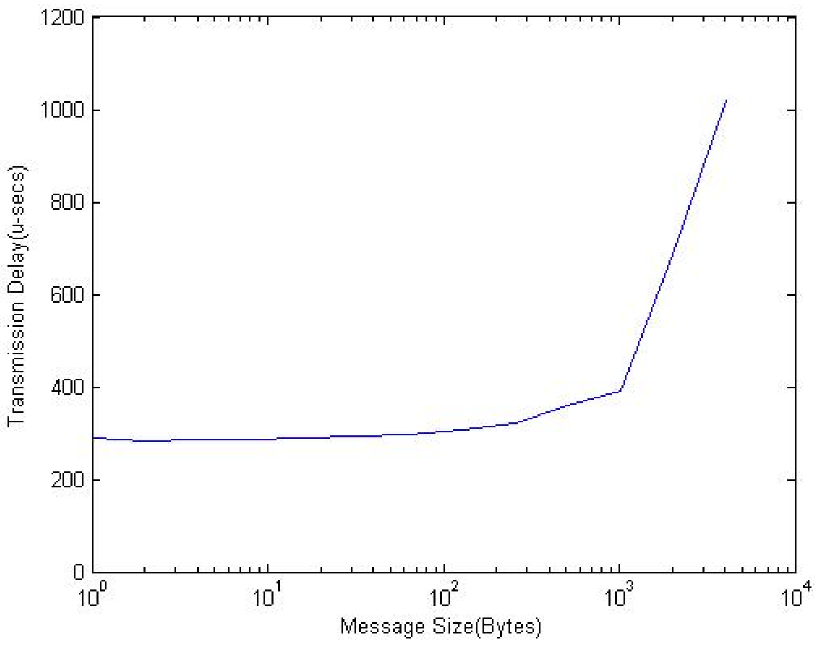
First, the timing properties of the communication medium in this implementation are the least influential. Given any medium (100Mbps Ethernet in this case) with dedicated hardware and well-defined message protocols, generating an estimate of how long it takes to physically move data across the medium is fairly straightforward with access to properties such as packet size, bandwidth, etc. On this platform, the transmission time was only readily measurable by the time interval starting when the communication library relinquishes control of a data message to the operating system to move onto the communication medium and ending when the "listen" routine of the receiver's communication controller determines a new message has been received. Every test has shown that this time interval is typically $70 - 80\mu$s; however, a conservative approach would be to assume 100μs.

On the sending side of a message transmission, the transmission delay was measured starting from the point the communication controller is awakened from a timeout for the next scheduled message transmission to the point where the communication library has successfully passed all of the message data to the operating system to put onto the communication medium. The converse is true for the reception delay of a message: it starts when the listening routine is notified a new message has arrived and ends when all of the updated message data has successfully been written into the shared memory of the receiving node. In summary, these values try to capture the entire message passing overhead that is introduced by both the operating system and the communication controller's implementation.

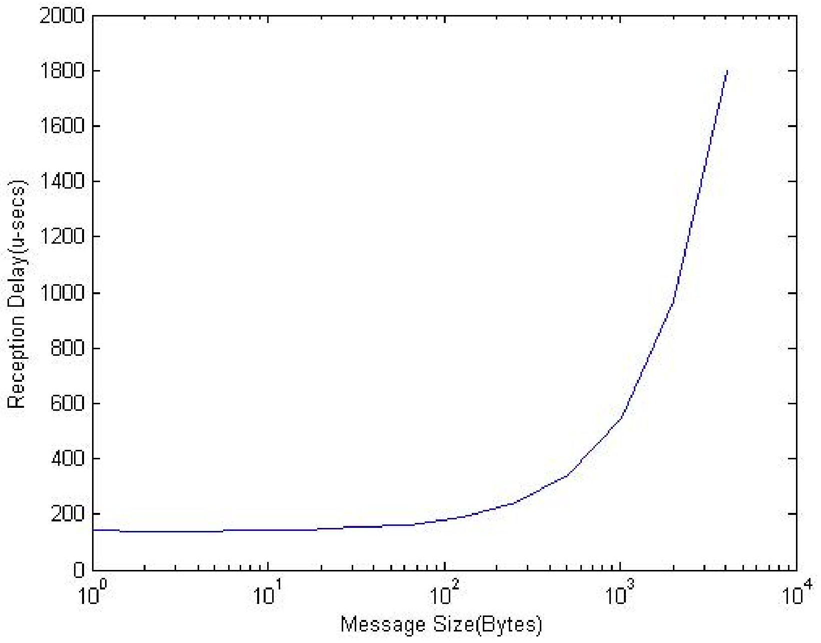
The following data (see Table 1 and Figure 12) was captured by testing a configuration of four nodes where each node was responsible for sending two messages of the same size each hyperperiod. Each test was conducted using four different configurations such that all four nodes assumed the role of master at least once. In total, the following values were determined from sending over 4000 messages per specified message size. These tests assumed an ideal scenario for uninterrupted communication where no periodic tasks were running and each message was scheduled 12ms apart from the next message (avoids any effects of the busy-wait implementation of the listening routine). Consequently, these results are a baseline or starting point for determining how closely messages can be scheduled provided a specific design.

Table 1: Message communication timing properties

Message Size(Bytes)	Transmission Delay(μs)	Reception Delay(μs)
1	288	141
2	283	139
4	287	139
8	287	142
16	290	146
32	292	153
64	295	163
128	306	192
256	321	241
512	359	342
1024	390	549
2048	686	970
4091	1021	1801



(a) Transmission delay



(b) Reception Delay

Figure 12: Message communication timing properties

Summary of Results for Communication Controller

The results from the above description of the communication controller provide the relevant timing properties that determine how closely a set of messages could be scheduled with relative confidence that messages would arrive at the receiver nodes before the new data is used by the TT tasks for calculations. From the results we can see that the use of large messages ($>$ 512 bytes) can very quickly require consecutive messages in the schedule to be separated by more than 1ms in order to have any certainty that collisions or insufficiently slow response times will not occur. Still, the variability of the timings that result from the use of Linux timing utilities (e.g. clock_nanosleep() and select()) dictate that it is not recommended to schedule any set of messages (regardless how small the message sizes are) with a separation interval smaller than 1ms. Furthermore, the reception delay values for the various message sizes must be taken into consideration when scheduling the TT tasks as well; the data in Table 1 must be used as a baseline for determining when a message must be transmitted so that the data is available to a task when it is released for execution.

Concerning the average timing properties: overshoot of timeout utilities (20μs), misalignment of hyperperiod start time between nodes (30μs), deviation from ideal hyperperiod intervals (40μs), hyperperiod synchronization lag between the communication controller and VM (20μs), and the message passing delays between nodes (see Table 1), the deviation from the ideal hyperperiod intervals is the least readily accounted for by either fixed offset adjustments or sufficient slack time between scheduled events. Currently, this deviation does accumulate over the operation of the system; therefore, the start of each hyperperiod will more than likely be misaligned from the ideal hyperperiod start time. This error would affect the controller performance since the accumulation effectively alters the period of execution of a control task (analogous to changing the sampling rate of a controller). Adaptive strategies in the communication controller and/or the tasks themselves could try to account for this; however, this underlying problem must be mitigated for this platform (in any of its potential hardware/software manifestations) to be applicable to safety-critical systems.

In conclusion, the presented implementation for the communication controller is capable of scheduling the periodic transmission of data messages over a shared communication bus with tens of microseconds accuracy. Granted that assumptions regarding the isolation (from external forces) of the network and the individual nodes seem idealistic, they are not atypical in the applications such an execution platform is intended for since it is often necessary to take as many precautions as possible to provide guarantees for reliability, robustness, and quality of performance. However, there are extensions and improvements to be made in order to provide a deployable TT communication controller.

FRODO: A Virtual Machine

In the control-oriented application domains previously discussed, TT tasks perform computations and environmental interactions in a strictly periodic fashion in order to maintain some level of desired system performance. This periodic execution model is extremely critical to these applications since the fundamental concepts and mathematical models of digital control theory typically assume this periodic behavior. Accordingly, any run-time system or VM that purports to support such applications must be able to precisely initiate the execution of periodic tasks according to a predefined schedule and provide other required interaction mechanisms (communication methods, synchronization primitives, etc.). The current implementation of the TT task scheduler (FRODO) is intended to provide these run-time mechanisms required by purely TT control applications; however, future extensions to FRODO will introduce other heterogeneous interaction mechanisms and task execution models (e.g. Event-Triggered MoC, remote-procedure calls, rendezvous, etc.).

The FRODO Task Model

Figure 5 and its respective logical description briefly laid out the fundamental points of the task execution model; however, we would like to further clarify all of the specifics in order to understand how applications have to be structured in order to use this execution framework.

At system initialization, the VM (FRODO) is provided the task schedule as an ordered list of all task execution instances that must occur in one hyperperiod. Each instance in the schedule will indicate the unique identifier of the task to execute and the scheduled release time (the *phase*) of the task instance relative to the start of the hyperperiod. Moreover, the system must be provided a header and a source file at compile-time that initialize the internal data structure for each task and provides the task's functional implementation (source-level instructions) respectively. Currently, the internal data structure supplements the descriptions from the task schedule with each task's respective (expected) WCET parameter.

With this information, the VM now knows when to schedule each task for execution relative to the start of the hyperperiod, where to find the source code/function implementation of each task, and when to stop a task from executing. The last remaining detail is how each task accesses the data (from shared memory) it must operate on to perform its calculations. Instead of having the VM handle all data transfers between tasks and shared memory, the current implementation allows each task to directly access data from shared memory in the same manner as the communication controller. Remember, this data is the same data that is encapsulated in the TT messages, and the shared memory structure is access-controlled (see Figure 7). This approach of removing the VM as an intermediary to shared memory was intended to both simplify the required task model (otherwise, all messages used as input or output to a task would have to be provided at compile-time) and reduce the overhead required to read/update data (tries to ensure timeliness of execution).

Run-Time Execution

During system initialization, the necessary steps for properly configuring the VM with the provided task information are performed and the POSIX thread for the VM is created. The system is always configured such that the communication controller thread has already begun executing when the VM thread starts. This ensures that the VM does not acquire the hyperperiod synchronization mutex before the communication controller. By doing so, the VM will immediately be blocked from executing until the first "HP_Start" signal is received from the communication controller (see discussion on synchronization of hyperperiod execution).

Upon successful synchronization with the communication controller, the VM must sequentially release each task for execution according to the task schedule using the same logical process that the communication controller uses for the message schedule. Fortunately, the run-time responsibilities of the VM are less restricting than those of the communication controller, i.e. the VM does not have to both schedule tasks and concurrently be listening for any other activity. Therefore, when the VM needs to schedule a task for execution by requesting a timeout, the VM can use the adjusted clock_nanosleep() routine instead of having use to select() or busy-waiting.

Whenever the adjusted clock_nanosleep() routine returns and the scheduled release time of the next task has been reached, the VM initiates the execution of the task's functional code. When a task needs to be released for execution, the VM creates a new POSIX thread that will be passed a pointer to the task's internal data structure and the VM halts its own execution (more on this below). When this new task thread executes, a generic function for the VM calls the task's implementation code using a predefined function pointer found in the task's data structure that was just passed as the thread's parameter. While the task's thread is executing, the VM thread has been halted using the adjusted clock_nanosleep() routine and should awaken at the expiration of the executing task's allowable execution time (calculated as the task's release time plus its WCET). Upon awakening from the timeout, the VM immediately guarantees the termination of the executing task by forced cancellation using the pthread_cancel() function, and the VM will either repeat the process for the next scheduled task or wait to receive the hyperperiod synchronization signal from the communication controller.

Unfortunately, the POSIX thread library supports only a limited number of usable utilities for dynamically controlling thread execution for TT applications (the wholesale creation of a new thread and forcibly cancelling a thread when its WCET is reached).

Future implementations will more than likely be able to exploit features common to more "real-time" capable frameworks that don't require such overhead-expensive operations.

Timing Properties of FRODO

Regardless of the inefficiencies of the current implementation and the use of POSIX threads, experimentation has shown that the VM is still capable of accurately scheduling a set of periodic tasks at the specified release times and terminating their execution when their WCET has been reached.

First, we would like to revisit the hyperperiod synchronization across the communication controller and VM threads. As previously mentioned, the current implementation yields an average lag time for the VM of $15 - 20\mu$s. This means the VM typically saves the hyperperiod start time slightly after the communication controller. This amount of error is very marginal considering the typical minimum time interval between scheduled tasks and messages is on the order of hundreds of microseconds if not more than a millisecond. Also, this lag could be accounted for by moving up the scheduled release time of tasks from the ideal time granted there is enough slack time available.

Concerning the accuracy with which task execution is initiated and terminated, both events are solely determined by the use of the offset-adjusted clock_nanosleep() routine. Accordingly, the effective release time of a task (from the viewpoint of when the VM creates the task's thread) is determined by how accurately the adjusted clock_nanosleep() routine awakens at a given timeout value, an average delay of 20μs. The same result also applies to the awakening of the VM for terminating an executing task once its WCET is reached. Another timing property that was important to capture was how long it takes the operating system to successfully terminate the task thread if it still running when the VM is awakened. Tests found that the POSIX utilities for forcibly cancelling a still executing thread added an additional overhead of 20μs.

Considering these results, the current VM implementation is very capable of scheduling a set of periodic control tasks with sufficient accuracy to meet the timing requirements for applications that operate with periods on the order of milliseconds. The non-ideal timing of the VM operations can all be accounted for by adjusting the task schedule (moving scheduled times earlier) if the average errors are assumed and there is sufficient slack time between scheduled events. Obviously, there will be limitations to how well this system can tolerate intermittent loading and interrupts during execution, but the current selection of "non-real-time" hardware and software does not yet undermine the capabilities of this system to support TT control applications.

Summary

This chapter presented the detailed specification of an execution platform capable of providing the run-time services of the TT MoC. The implementation used only readily available hardware devices and the standard Linux operating system, and the operational logic responsible for maintaining the periodic execution of both TT message and task schedules was extracted from the executable DEVS specification of the TT MoC provided in the previous chapter. This realization of the execution platform successfully illustrates how the DEVS modeling framework can facilitate the operational specification and rapid prototyping of software systems. Motivated by the TTA, the resulting node-level implementation architecture consists of a communication controller, a task scheduler or VM, the individual software tasks, and a shared memory structure that allows the exchange of data elements between the tasks and the communication protocol (similar to the CNI of TTA). Lastly, a multi-node system can be built upon any communication medium that supports a shared bus configuration for broadcast communication.

Although the platform can meet behavioral expectations of TT systems by logically executing the periodic schedules, the use of off-the-shelf hardware and software components introduces temporal inaccuracies during execution that would result in suboptimal performance in most real-time systems. The discovered timing errors and adjustment factors specific to this implementation are provided in Table 2. Even though the errors are non-ideal, many of them can be corrected or disregarded by adjusting the schedules to align events at the desired time and ensuring there is enough slack time (when it is available) in between scheduled events. Unfortunately, the inter-hyperperiod interval error is not readily corrected without substituting hardware or extending the implementation with more complex synchronization algorithms. Regardless of the method, improving the hyperperiod synchronization at the system-, node-, and thread-level could greatly improve the performance and reliability of this platform. Taking into consideration the given timing properties, it is recommended that all scheduled events (task release, task termination, message transmission, and hyperperiod synchronization) should be separated by at least 1ms; more time should be added if it involves the transmission of large data messages.

The timing errors and properties in Table 2 help system designers construct realistic expectations of how a system will execute on the physical implementation. Also, they can be reintegrated into the DEVS model of the MoC. In the original DEVS model of the TT MoC (see Figure 7), many of the lifespan functions of states were specified with the knowledge that any concrete platform implementation would introduce delays and timing inconsistencies; however, providing specific values or distributions for the lifespans was unreasonable without having a reference implementation. Now equipped with the current implementation and its timing characteristics, the lifespans of the DEVS model can be updated to match these parameters/variances in order to improve the accuracy of the model simulations when compared against the physical execution platform. This will be illustrated in the following example.

In the following chapter, the envisioned design process that utilizes the DEVS modeling framework to develop applications for the above execution platform is presented. A simple example application used in various real-time systems is modeled and simulated in DEVS and then deployed on the execution platform. The resulting execution traces from DEVS and the platform are compared to determine if the updated DEVS model can produce simulation traces temporally equivalent to the run-time execution.

Table 2: Summary of timing properties for current implementation

Description	**Value(μs)**
Adjusted clock_nanosleep() error	20
Inter-hyperperiod interval error	40
Intra-hyperperiod node sync. error	30
Transmission delay of hyperperiod sync. message	250
VM hyperperiod sync. error	20
Timeout for VM sync.	350
Busy-wait period for select()	400
Transmission delay over Ethernet	100
Shared memory to bus delay	>280 (see Table 1)
Bus to shared memory delay	>140 (see Table 1)
Releasing a TT task error	20
Terminating a TT task error	40

CHAPTER V

USE CASE: TRIPLE-MODULAR REDUNDANCY

As a guiding example, we show how to use the DEVS modeling framework in conjunction with the FRODO execution platform to prototype, deploy, and evaluate (for correctness) a common software design strategy for distributed real-time systems. A representative DEVS model of a distributed real-time system will be constructed, and the DEVS simulation engine will enable us to evaluate if the given implementation model (task and message schedules) produces an execution trace indicative of the desired system operation. The modeled system will then be deployed on the FRODO platform and supplemented with the executable task code to see if the platform execution can discernibly match the execution trace generated from the DEVS engine.

Triple-Modular Redundancy

Some level of resilience to faults in the value domain is a necessity in any fault-tolerant real-time system. Since all data elements present in these systems are directly affected by hardware (e.g. sensors, communication infrastructure, and processors), the presence of a hardware fault will most certainly affect the system performance if it goes unnoticed and uncorrected. For many decades, this potential pitfall has motivated the use of redundant hardware units to boost the trustworthiness of operational data. However, comparing data from two sources alone cannot solve the problem (which one to choose?); therefore, a minimum of three independent hardware units is required to provide any assurance of data integrity.

Determining data consensus from three hardware units is called Triple-Modular Redundancy (TMR) [3]. A TMR implementation further requires a fourth independent entity, the *voter*, to perform the comparison between the three data values to establish the majority agreement. Two common techniques used by a voter to establish a data agreement are *exact* and *inexact* voting: data values must be either identical or within a tolerated difference interval respectively. After a result has been determined, it is passed on to memory or some other application-level component, and the process repeats with the next available data set. Various other strategies can be employed for taking corrective action if one of the sources continually sends fault-indicative data or if all three sources produce three distinct data values.

Experimental Description

In order to construct a system model, we must first gather the system components and establish how they restrict the implementation. First, we will once again assume the system configuration of Figure 9: four Soekris net4801 embedded boards that communicate using the UDP protocol over an isolated 100 Mbps Ethernet network. In this experiment, a physical sensing device is attached to three of the nodes (as the configuration indicates) and the fourth node functions as the voter. The chosen sensor is the Honeywell HMR3300 digital compass. Digital compasses are common instruments in distributed control applications, especially aviation systems. These systems obtain physical orientation information from such sensors (the HMR3330 provides directional heading, roll, and pitch) to facilitate the precise positioning and tracking of moving objects. Each node equipped with a compass will read in the sensor data and extract the data component of interest (directional heading in this case), and the voter application on the fourth node will determine the majority agreement between the three data values.

The timing characteristics of the FRODO platform on the Soekris boards have been determined, so we must now consider how the use of the HMR3300 impacts the system scheduling. As a time-triggered system, the rate at which the sensor provides data will have the greatest effect on how frequently the TMR application must execute to evaluate the data set. The HMR3300 documentation indicates the sensor's data update rate is 8 Hz, the maximum communication speed is 19200 Baud over a serial link (UART), the maximum length of the output data string from one sensor reading is 20 bytes, and the serial protocol requires 10 bits to communicate one data byte (1 start bit, 8 data bits, 1 stop bit). Also, the directional heading (in degrees) will be the first entry in the output string of the sensor with possible values 0.0 – 359.9 (it is separated from the next entry by a comma). Consequently, the calculated worst-case time interval between consecutive sensor readings is 135.42ms (see equation below). This value will determine the period and WCET for tasks that collect the sensor readings as well as the hyperperiod of the task and message schedules.

$$\frac{1}{8}s + \left(\frac{20\ bytes}{1}\right)\left(\frac{10\ bits}{1\ byte}\right)\left(\frac{1\ s}{19200\ bits}\right) = 135.42ms \tag{2}$$

<u>DEVS Model and Simulation of a TMR System</u>

A DEVS model of the TMR system is not intended to capture the operational logic implemented by the TT tasks; instead, the DEVS model enables the developer to evaluate how a provided set of message and task schedules results in a timed event trace through simulation. Given that the DEVS model of the TT MoC is detailed with state lifespan functions that capture the timing variability of the physical execution platform (e.g. hyperperiod synchronization, message/task scheduling, data-dependent message transmission, etc.), the produced event traces can illustrate to the developer possible deviations from the desired scheduled execution that could result in message collisions, missed data messages, or other undesirable behaviors.

Accordingly, message and task schedules for the implementation of the TMR compass example must be constructed considering the timing information derived from the compass documentation and the known system configuration. First, the system configuration and the experimental description make the task and message allocation straightforward: all four nodes will have one TT task (three will gather data from a compass and the fourth will act as the voter) and each node will send one message per hyperperiod (the three sensor nodes will each send a message containing their latest compass reading and the voter will send a message containing the data value agreed upon by the TMR evaluation). Regarding the scheduling of the tasks and messages, the calculated worst-case time interval between sensor readings (135.42ms) has the greatest influence over the schedules since this value determines how often a new set of data values is available for the voter to evaluate according to the chosen TMR agreement criterion. Common practice of TT real-time systems dictates this worst-case time interval parameter should be used to set the WCET of the three sensor reading tasks—a value of 137ms was chosen to give plenty of slack time in case of timing inconsistencies. Luckily, the WCET parameter of the TMR voter task can be much shorter since it merely performs a quick comparison of the data set—2ms was chosen. Concerning the ordered execution of tasks, the three sensor tasks can execute simultaneously (on independent nodes) since they read independent sensors, and the voter task will execute shortly thereafter since it operates on the most recent sensor readings.

The last scheduling decision is how to fit the messages between the completed sensor readings and the TMR task execution. Obviously, the three sensor data messages must be serialized since they cannot share the bus. Considering this type of application is common to control systems that desire the least amount of latency between a sensor reading and the resultant output action, it would be best to schedule the messages as closely as possible without the potential for collisions. Considering the recommendations from the description of the execution platform to schedule messages at least 1ms apart, the conservative approach was taken here by allowing 1.5ms between message transmissions. The extra time was included to account for possibly slow response times since the receivers would be in the busy-wait phase of listening to the bus (see description of TT communication controller implementation). Furthermore, the delay between the completion of the first sensor reading task and its corresponding message transmission (1.5ms) is larger than the delay between the completion of the TMR voter task and its respective data message (0.5ms), since the likelihood of the sensor reading task expending its WCET is much higher than the TMR voter task doing the same (the former has to interact with hardware to read data whereas the latter performs a simple data comparison). Taking into account these timing characteristics, the hyperperiod for both the message and task schedules was selected to be 150ms. The corresponding message and task schedules are indicated in Table 3 and their ideal event trace is illustrated in Figure 13 (the time axis is in milliseconds). Note the 1ms interval between the transmission of message "m3" and the start of execution for the TMR voter task. This shortened time frame was intentional and will be used to illustrate the varying timed behaviors that can occur on the FRODO platform (see Figures 14(a) – (c)).

Table 3: Scheduled messages and tasks for TMR compass example (hyperperiod of 150ms)

Node ID	Messages ID: (Phase(ms), Size(bytes))	Tasks ID: (Phase(ms), WCET(ms))
C1	m1: (139.0, 5)	sensor_COMPASS_1: (0.5, 137.0)
C2	m2: (140.5, 5)	sensor_COMPASS_2: (0.5, 137.0)
C3	m3: (142.0, 5)	sensor_COMPASS_3: (0.5, 137.0)
C4	m4: (145.5, 5)	compute_TMR: (143.0, 2.0)

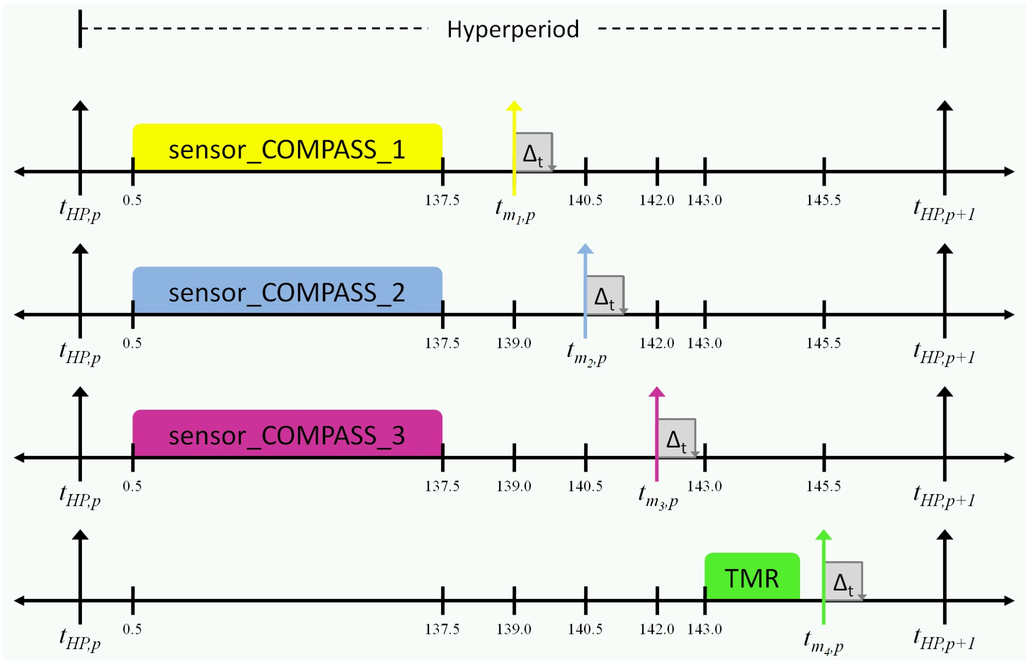


Figure 13: Timeline of events for TMR compass example

Given these schedules and the ideal-case event traces, we can now examine how the DEVS simulation of the modeled execution framework in Figure 7 introduces timing inconsistencies to the event traces that are indicative of potential behaviors on the implementation of the TT execution platform. If the schedules were created without proper consideration, the simulation may illustrate how deteriorated controller performance if not outright failures could occur.

The output below (see Table 4) is a segment of the DEVS simulation trace (generated using DEVS++ and the above specifications of the message schedule, task schedule, and a listing of the four controllers). Any time a discrete transition (internal or external) occurs throughout the simulation, the DEVS simulator logs the transition along with the pre- and post-transition state information (not shown) for the entire coupled model. Each line in the listing below is one such transition that matches with the events shown in the in timeline of Figure 13. The last segment of each transition line indicates the time of occurrence for the transition in milliseconds (e.g. the first transition specifies that the communication controller of node "C2" synchronized the start of the hyperperiod with its VM at a time 0.007ms after the start of execution). Following down with the rest of node "C2" transitions: it released its sensor reading task at 0.527ms, the task completed executing at 137.472ms (slightly before its WCET), it received message "m1" at 139.526ms, it transmitted its message "m2" at 140.529, and it started its next hyperperiod at 150.044ms. All of the scheduled events for "C2" occurred later in the simulation than compared to the ideal trace; however, none of these errors introduced a discrepancy in execution that would necessarily lead to a failure.

Implementation of TMR on FRODO

Getting the TMR compass application running on the Soekris boards equipped with the FRODO execution platform required the same schedules and controller configuration file as the DEVS simulation along with the actual source code for the tasks. Without getting into great detail, it is not hard to imagine that the code was fairly straightforward (especially considering three tasks perform the same function). The sensor reading tasks had to open a communication link to the sensors using the serial port of the Soekris boards and a Linux file descriptor, and upon acquiring each new sensor reading, the tasks had to parse the data in order to extract the directional heading value from the string (the bytes leading up to the first comma). Lastly, the newly obtained heading value was written into the shared memory location of the appropriate message ("m1", "m2", or "m3" depending on which node) to later be transmitted by the communication

Table 4: Segment of DEVS++ simulation output for TMR compass example

```
//First synchronization of hyperperiods across the four nodes
({!C2.CC.C2_HPS,?C2.VM.C2_HPS},t_c=0.007)-->
({!C4.CC.C4_HPS,?C4.VM.C4_HPS},t_c=0.011)-->
({!C3.CC.C3_HPS,?C3.VM.C3_HPS},t_c=0.014)-->
({!C1.CC.C1_HPS,?C1.VM.C1_HPS},t_c=0.016)-->

//Release of the three sensor reading tasks
({!C2.VM.sensor_COMPASS_2_REL,?C2.sensor_COMPASS_2.rel},t_c=0.527)-->
({!C3.VM.sensor_COMPASS_3_REL,?C3.sensor_COMPASS_3.rel},t_c=0.529)-->
({!C1.VM.sensor_COMPASS_1_REL,?C1.sensor_COMPASS_1.rel},t_c=0.534)-->

//Completion of sensor reading tasks
//note: this is self-completion and not forced by the VM
({!C2.sensor_COMPASS_2.fin,?C2.VM.sensor_COMPASS_2_FIN},t_c=137.472)-->
({!C3.sensor_COMPASS_3.fin,?C3.VM.sensor_COMPASS_3_FIN},t_c=137.475)-->
({!C1.sensor_COMPASS_1.fin,?C1.VM.sensor_COMPASS_1_FIN},t_c=137.485)-->

//Transmission of message ``m1''
({!C1.CC.C1_Send:(m1,5),?Bus.TTBus.0_busRec:(m1,5)},t_c=139.032)-->

//Receipt of message ``m1''
({!Bus.TTBus.1_busSend:m1,?C2.CC.C2_Rec:m1},t_c=139.526)-->
({!Bus.TTBus.2_busSend:m1,?C3.CC.C3_Rec:m1},t_c=139.526)-->
({!Bus.TTBus.3_busSend:m1,?C4.CC.C4_Rec:m1},t_c=139.526)-->

//Transmission of message ``m2'' and ``m3''
({!C2.CC.C2_Send:(m2,5),?Bus.TTBus.1_busRec:(m2,5)},t_c=140.529)-->
({!C3.CC.C3_Send:(m3,5),?Bus.TTBus.2_busRec:(m3,5)},t_c=142.021)-->

//Release and completion of TMR voter task
({!C4.VM.compute_TMR_REL,?C4.compute_TMR.rel},t_c=143.03)-->
({!C4.compute_TMR.fin,?C4.VM.compute_TMR_FIN},t_c=144.977)-->

//Transmission of ``m4''
({!C4.CC.C4_Send:(m4,5),?Bus.TTBus.3_busRec:(m4,5)},t_c=145.518)-->

//Second synchronization of hyperperiods
({!C4.CC.C4_HPS,?C4.VM.C4_HPS},t_c=150.030)-->
({!C3.CC.C3_HPS,?C3.VM.C3_HPS},t_c=150.043)-->
({!C2.CC.C2_HPS,?C2.VM.C2_HPS},t_c=150.044)-->
({!C1.CC.C1_HPS,?C1.VM.C1_HPS},t_c=150.053)-->
```

controller. The TMR voter task first retrieves all three sensor readings from its shared memory, and then applies the TMR evaluation function to determine the consensus on the current heading of the system. The agreed upon data value is then written into the shared memory location of message “m4” to later be transmitted. In this example, an *inexact* evaluation function was used.

Table 5: FRODO output

m1: 074.0
m2: 074.1
m3: 074.1
TMR: 74
m1: 074.2
m2: 074.1
m3: 074.2
TMR: 74
m1: 070.5
m2: 074.2
m3: 074.2
TMR: 74
m1: 079.9
m2: 074.2
m3: 074.1
TMR: 74
m1: 087.7
m2: 074.1
m3: 074.1
TMR: 74
m1: 087.7
m2: 074.1
m3: 074.1
TMR: 74

Table 5 is a segment of the run-time data (the output of the executing tasks) as it goes through the tasks for the TMR agreement evaluation. Each set of four lines lists the three sensor readings for the directional heading (“m1”, “m2”, and “m3”) followed by the result of the TMR voter task performing the evaluation (“m4”). This implementation uses a very unintelligent version of *inexact* voting: any two sensor readings with the same integer value form a majority consensus, i.e. the decimal point values are ignored, and the previous consensus value is taken if all three sensor readings disagree. From the run-time data of Table 5, we can see that the first two samples of the sensors produced a three-way agreement (a heading of 74 degrees), and then this is followed by a series of sporadic values for “m1”, yet “m2” and “m3” continue to maintain a consensus of 74 degrees. This very simple but practical example is meant to supplement the results presented in Figure 15(b) that illustrate this implementation of the TT execution platform can produce tractably correct execution results/traces in both the temporal and value domains.

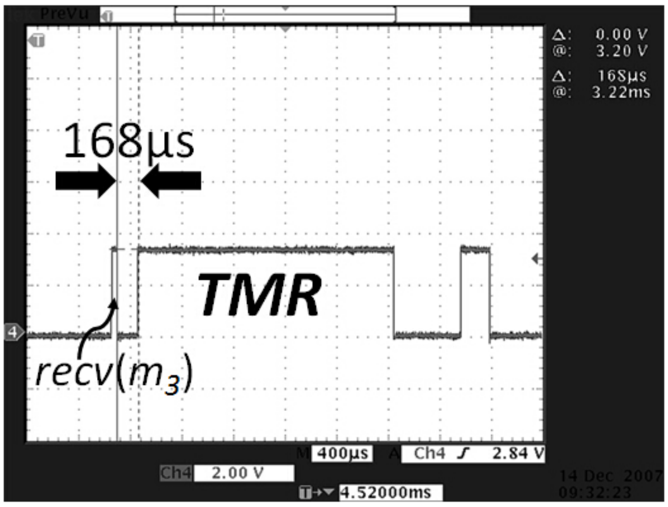
While previously determining the message and task schedules, it was indicated that the time interval between the transmission of message “m3” and the execution of the TMR voter task was only 1ms instead of 1.5ms. This was done to illustrate the variability in the system and why the choice of 1.5ms for the other event spacing intervals was a motivated selection.

Figures 14(a)–(c) are samples that were taken during execution of the TMR system; each one captures a different point at which the communication controller of node “C4” received message “m3” and had updated the contents of shared memory. The first, short pulse in the left of each output indicates the receiving of the message, and the second, longer pulse is the execution of the TMR voter task. The scope's cursors were used to measure the time interval between receiving the message and starting the execution of the task. These three samples (168μs, 360μs, and 592μs) are indicative of the typical values that continually appeared while running this experiment.

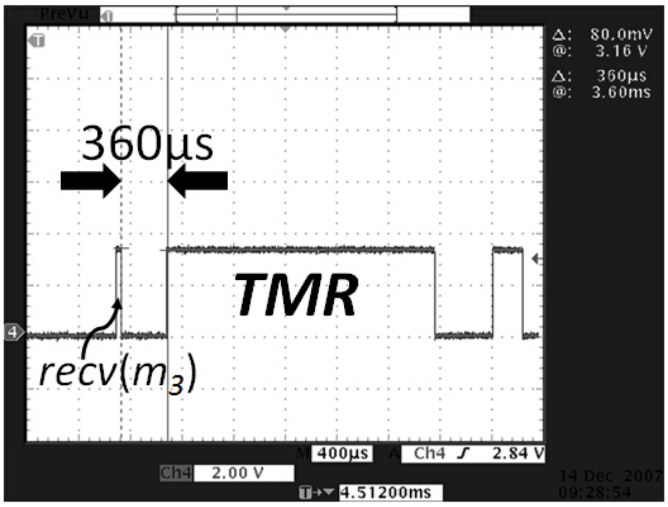
This wide range of time intervals can be explained by the busy-wait phase of the TT communication controller's “listening” routine. Since node “C4” will be transmitting a message less than 8ms after the transmission of message “m3” (3.5ms to be exact), the use of the select() function is limited to busy-waiting due to the “jiffy” signal problem. Unfortunately, when message “m3” is transmitted and reaches “C4”, there is no way of knowing where in a busy-wait cycle node “C4” is halted; therefore, one cannot explicitly predict how long before “C4” will respond to the new message. Given that the current busy-wait period is 400μs, it is not surprising that the extreme cases of the response time interval are near 400μs apart (the other excess time can be attributed to inaccuracies in the adjusted clock_nanosleep routine and overhead of returning from function calls in the implementation). Once again, the inaccessibility to low-level timing and communication infrastructure of the hardware restricts the ability to schedule messages and tasks very closely; however, using a time interval of at least 1ms (if not 1.5ms) is not restricting enough to make the FRODO execution platform unusable to TT systems.

Comparing Results of DEVS and FRODO

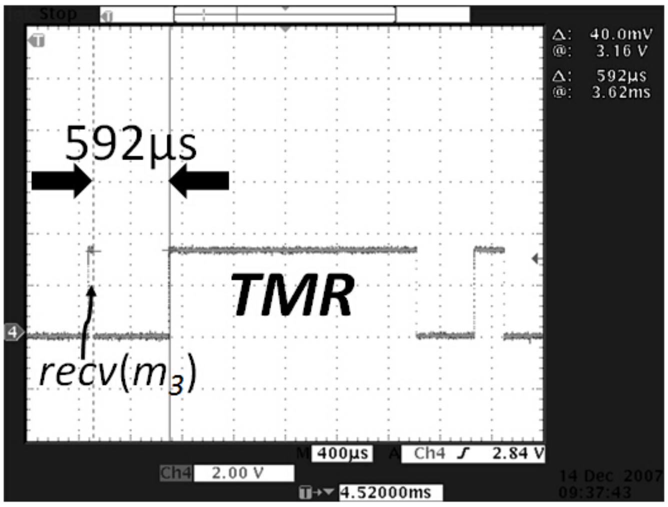
Figure 15(a) and Figure 15(b) show the corresponding execution traces of the TMR example from the DEVS and FRODO execution frameworks respectively. Due to the resolution of the oscilloscope (used for the FRODO trace) and the large hyperperiod,



(a) Short(168μs)



(b) Medium(360μs)



(c) Long(592μs)

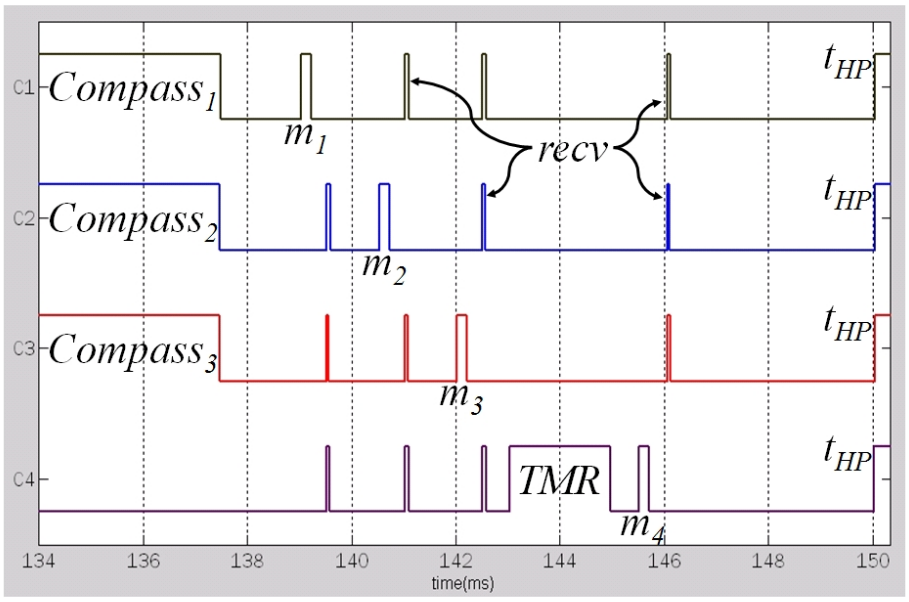
Figure 14: Varying response times to message “m3” on node “C4”

the trace segment only captures the activity directly after the completion of the senor reading tasks leading up to the next hyperperiod synchronization point (the output starts 134ms after the previous hyperperiod synchronization). The events of greatest interest are labeled in both traces: message transmission (m_{1-4}), task execution ($Compass_{1-3}$ andTMR), and the hyperperiod synchronization (t_{HP}). Furthermore, the narrow pulses that appear on each channel directly following the transmission of a message are the other nodes indicating the completed reception of the preceding message (see *recv*). This verifies that the message is received and properly stored in shared memory before any of the subsequent tasks (such as the TMR voter) require the message data.

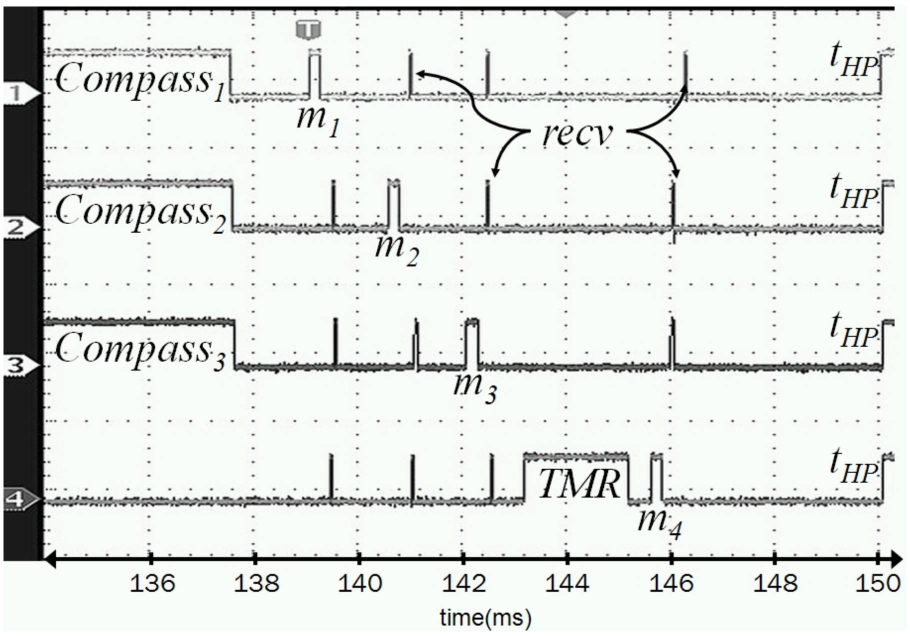
Firstly, it is important to note that the output in Figure 15(b) verifies that the physical implementation of the TT MoC is successfully scheduling the transmission of messages and the execution of tasks such that the desired timed event traces are produced (with a quantifiable degree of error). This current implementation on off-the-shelf hardware and software without "real-time" capabilities can effectively schedule events with tens of microseconds accuracy. Secondly, comparing this output with the DEVS execution trace (Figure 15(a)) shows that the updated DEVS model of the TT MoC can produce run-time simulations of the same system with similar timing errors/variances. Recall, this was achieved by simply incorporating the performance data gathered from the implementation into the lifespan functions of discrete states in the DEVS specification of the TT MoC.

Summary

In this chapter we used a common software approach to providing a level of value domain fault-tolerance in distributed real-time systems to demonstrate the presented system design strategy of using the DEVS modeling framework paired with a TT execution platform. A simple application that provided Triple-Modular Redundancy for reading sensor data was developed by first considering the timing effects introduced by the selected sensor hardware, then prototyping a possible implementation with the DEVS modeling framework to evaluate possible event traces derived from the generated execution schedules and the DEVS model of the TT MoC, and finally, deployed on the physical TT



(a) DEVS



(b) FRODO

Figure 15: Execution traces of TMR example

execution platform to determine if the true run-time performance of the system matched expectations.

Unfortunately, the results did corroborate the previous concerns that timing variances would be exhibited in the execution traces due to the "non-real-time" implementation of the execution platform; however, the errors are not completely unaccounted for in the current implementation and potential extensions could further mitigate their effects. Also, the isolated operating conditions of the execution platform are preventative measures that further restrict the likelihood of the timing variances becoming unbounded or measurably worse than the figures obtained during testing and benchmarking.

The same "non-ideal" operation confirmed that the generated simulation runs from the DEVS modeling framework are comparable to the execution traces of the timed events from the execution platform. Furthermore, the operational (data domain) behavior of the executing system was confirmed as well, thereby supporting the objective of constructing a completely TT execution platform capable of providing time-deterministic system execution using off-the-shelf hardware and software components.

CHAPTER VI

CONCLUSION

This thesis introduced a modeling framework for describing the operational semantics of Models of Computation. Models of computation are specified using the DEVS modeling formalism [10, 11] in order to capture the run-time logic that initiates the execution and interactions of software components. The selection of DEVS was a calculated choice motivated by the similarities between the underlying semantics (functions) that describe the state evolution of a DEVS model and the classes of behaviors common to embedded systems: event-triggered and time-triggered. Consequently, DEVS is a viable client for exploring the effects of composing heterogeneous models of computation to yield more complex behavioral and interaction patterns. Modeling such challenging (yet very realistic) software systems will not obfuscate the inner workings of model execution under incomprehensible logic or notation, but will illustrate to a developer the consequences of intermixing behavioral/interaction types in the same natural reasoning the systems are originally conceptualized, i.e. operationally.

To supplement the description of the modeling framework, an example showed how a DEVS representation of a model of computation can be implemented on readily available hardware and software. The physical realization of the model provides an execution platform intended to execute software components according to the model of computation. Specifically, a DEVS model of the time-triggered model of computation served as a blueprint for implementing the necessary platform logic, and the resulting system successfully performed the periodic execution of message and task schedules with marginal timing errors.

This exercise of capturing the logic in DEVS before developing the physical platform greatly reduced the mental effort required to realize the execution logic of the time-triggered system. Instead, the challenges faced during implementation arose when attempting to achieve the accurate scheduling of events using hardware and software components not intended for timing-precision or real-time behavior. Bounded and predictable timing results were achieved by strictly isolating the individual nodes and the communication network from external event/interrupt sources (a common assumption in safety-critical systems).

These results are useful for reasoning about run-time performance and the limitations of scheduling tasks and messages on the platform. Also, the same data was integrated into the DEVS model of the time-triggered model of computation (as updated lifespans) in order to reflect the limitations of the platform. Using the updated model, a simple real-time application was developed using the modeling framework and then implemented on the platform. The execution results from each were then compared to see if the DEVS simulation could reasonably match the run-time behavior of the physical implementation. Analysis proved that indeed the execution traces of the DEVS model introduced timing variances and delays in the occurrences of scheduled events that were similar to the inaccuracies of the physical implementation. By introducing these implementation-specific details, the DEVS framework empowers the developer to reason about a design and its physical limitations without having the platform or providing the application-level code.

Future Work

Regarding the platform implementation, it could most obviously benefit from improvements in the algorithms and techniques used to synchronize the scheduled events across the individual nodes. Instead of adjusting the offsets and errors that result from the limitations of Linux, it would be more effective and fitting of real-time systems if a robust approach was taken that would synchronize the clocks and/or account for clock drift on each node. Still, the current use of the standard Linux operating system is non-ideal since it provides no guarantees regarding timing and manipulating/accessing low-level hardware for timing and message handling is very cumbersome. Porting the implementation to more "real-time" capable operating systems has always been planned (and is outright necessary to meet strict real-time requirements); therefore, this implementation was developed with as few as possible Linux-specific constructs and techniques. Ideally, the effort required to migrate the implementation to other platforms will be minimal.

The DEVS modeling framework presents plenty of opportunities for potential growth in uses and adaptations. First, more models of computation with richer behavioral sets need to be specified in order to evaluate the usefulness and capabilities of this approach. With more behavioral categories and experience, the possibility to compose heterogeneous models of computation becomes a possibility as well. The ability to model and operationally interpret the consequences of intermixing behaviors and interactions will be extremely useful, and the approach presented here seems to be very conducive to tractably following the effects of such compositions. Still, with more use and complex behaviors, the need to analyze DEVS models and execution traces in a mathematically rigorous framework will become apparent. This framework and DEVS in general do not support such techniques [9]; therefore, investigating other reasoning frameworks and analysis tools that would readily support the reinterpretation of DEVS models into their native syntax would be worthwhile.

BIBLIOGRAPHY

[1] Thomas Henzinger, Benjamin Horowitz, and Christoph Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, January 2003.

[2] Moon Ho Hwang. *DEVS++: C++ Open Source Library of DEVS Formalism.* http://odevspp.sourceforge.net/, first edition, May 2007.

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

[4] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, Jan 2003.

[5] Hermann Kopetz and Günter Grünsteidl. TTP - A protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, 1994.

[6] Hermann Kopetz, Michael Holzmann, and Wilfried Elmenreich. A universal smart transducer interface: TTP/A. *International Journal of Computer System Science & Engineering*, 16(2), Mar. 2001.

[7] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. A denotational framework for comparing models of computation. Technical Report UCB/ERL M97/11, EECS Department, University of California, Berkeley, 1997.

[8] Jim Naruto. *Adevs (A Discrete EVent System simulator) C++ Library.* http://www.ornl.gov/∼1qn/adevs/index.html, 2008.

[9] Hans Vangheluwe. Personal communication with Dr. Gábor Karsai, 2006.

[10] Bernard P. Zeigler. *Theory of Modeling and Simulation.* John Wiley, 1976.

[11] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation.* Academic Press, Inc., Orlando, FL, USA, 2000.

[12] Bernard P. Zeigler and Hessam S. Sarjoughian. *Introduction to DEVS Modeling and Simulation with JAVA. DEVSJAVA Manual.* http://odevspp.sourceforge.net/, 2003.