TIME-TRIGGERED HIGH-CONFIDENCE EMBEDDED SYSTEMS: MODELING, SIMULATION, ANALYSIS AND BACK

By

GRAHAM S. HEMINGWAY

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

May, 2011

Nashville, Tennessee

Approved

Janos Sztipanovits

Gabor Karsai

Xenofon Koutsoukos

Joseph Sifakis

Larry Schumaker

DEDICATION

This thesis is dedicated to my loving family: Celeste, Penny, Mom, Dad, and Kristen. Without your encouragement and support, this work would have been neither possible nor fun. I can never express my gratitude enough.

ACKNOWLEDGEMENTS

Portions of this work were sponsored by the Air Force Office of Scientific Research (AFOSR), USAF, under grant/contract number FA9550-06-0312. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government. I would like to thank the AFOSR for their support over the years.

I would also like to acknowledge the superlative guidance of my advisor, Professor Janos Sztipanovits. He was willing to welcome an untested and green graduate student into his hectic schedule. Over the years I have spent with him, I feel that I have become a part of his amazing family both in work and in life. Thank you, Professor, for all of the opportunities and support you have given me.

Finally, I must acknowledge my research partners, teammates, and leadership: Joe Porter, Nicholas Kottenstette, Harmon Nine, Chris van Buskirk and Gabor Karsai - thank you for all of the help and direction you have given me through the years we have worked together. I feel, as I hope that you do, that the fruits of our combined efforts have far exceeded what any one of us could yield alone.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER I

INTRODUCTION

High-Confidence Embedded Systems

Cyber-physical systems (CPS) are composed of embedded software components interacting with their environment through hardware-based sensors and actuators [2], as illustrated in Figure 1. These systems have become a pervasive and fundamental aspect of much of the infrastructure in the world around us. As humans rely upon these systems for more critical tasks, understanding their behavior becomes ever more important. In situations where a system failure leads to significant impact or loss of life, such as flight control systems or pacemakers, we must have high confidence about the properties and capabilities of a system's design.

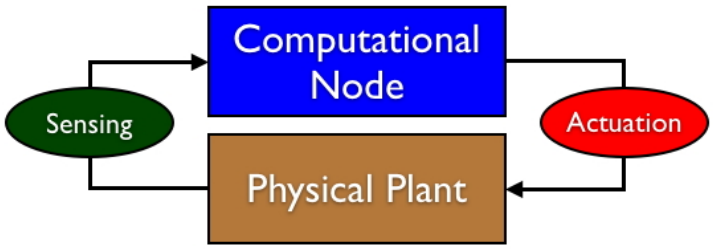

Figure 1: Cyber-Physical System (CPS)

The task of developing safety-critical embedded systems can be daunting. The burgeoning scale and complexity of many such systems makes fully determining their behavior problematic. Systems-of-systems, large-scale compositions of smaller systems, are particularly complex and challenging. Existing methodologies and tools for CPS development enforce iterative processes which naturally lead to frequent rework of both the software and hardware components of a design. Each iteration requires revalidation of system behaviors either through analysis or simulation. While construction of complete prototypes allows for thorough testing and analysis, the cost associated with iteratively building prototypes makes this approach untenable, especially as systems scale ever larger. Purely

analytic validation methods are making rapid progress [3], but are not always capable of verifying complex designs, let alone large-scale systems-of-systems.

Numerous formalisms exist to facilitate designing safety-critical systems. The time-triggered (TT) model of computation has demonstrated itself to be a robust execution model for high-confidence distributed systems [4, 5]. While certainly not the only viable model of computation for safety-critical systems, in theory, it provides robust timing and fault detection characteristics. The trade-off is that time-triggered execution places constraints on the software components running within it, such as communication times, execution durations, and inter-node synchrony. With a strong TT abstraction, lower-level platform details, such as operating system, memory, processor speed, caching, etc. can be ignored as long as those details do not invalidate the fundamental time-triggered assumptions.

Model driven development (MDD) has emerged as an approach for CPS development that simplifies the process of designing complex systems by abstracting their design using digital models [6, 7]. This higher level of abstraction lends itself to useful behavioral analysis while lower-level details, such as executable code, can easily be synthesized from the model. Design iterations still occur in MDD but tend to be less frequent as compared to low-level design and also tend to require less manual rework since many artifacts are automatically derived from the model. While MDD provides many tools for easing the development of complex systems the challenge remains to create a toolchain that allows designers to constructively develop high-confidence systems while also offering a means to understand their behavior.

There remain numerous challenges in creating a toolchain for time-triggered high-confidence embedded system design using an MDD approach. These challenges include:

- **Consistent & Formal Definition of Time-Triggered Systems.** The concept of a schedule-driven execution model is straightforward to understand, but much more complex to capture in a design specification. No broadly accepted definition of a time-triggered system is found in literature or in practice. Numerous research efforts have tied time-triggered behavior to other models of computations, such as discrete-event systems or timed automata, but these abstractions can not fully capture all of the nuances of time-triggered system designs. The

lack of a consistent system definition reduces model and tool interoperability and increases the workload burden on system designers.

- **Time-Triggered Platform Realization.** The time-triggered approach has many theoretical advantages, but realizing an executable time-triggered platform is challenging. Our interpretation of a TT platform demands that both task execution and message communication must conform to a statically determined schedule. High-confidence controllers frequently operate above 100Hz, thus requiring high frequency execution and minimal tolerance for timing inaccuracies, all while checking for possible error conditions. A further challenge is to realize consistent time-triggered execution in both resource-constrained (e.g. RAM, CPU capacity, channel bandwidth) environments and across a range of computing platforms.

- **Platform Effects Modeling.** The behavior of a time-invariant software controller can differ significantly from that of the same controller deployed onto actual hardware, especially distributed architectures. The deployment of software components onto a hardware platform, with its associated CPU, memory, communications channel, etc., frequently alters the expected behavior of the controller. Time-triggered execution helps to mitigate some platform effects, but many issues are only found once the hardware and software have been combined and the final system is executed.

- **Distributed & Heterogeneous Simulation.** A typical modern automobile can contain over 80 embedded processors and three separate communication networks, perhaps more [8]. Existing tools, such as Matlab/Simulink [9], are able to simulate the execution of a single controller or a small collection of controllers, but neither scale well nor allow for distributed processing. Additionally, simulation of a complex embedded system may require the heterogeneous composition of many smaller simulation tools in order to form a more accurate view of the whole system. Creation of such large-scale, distributed and heterogeneous simulations is both error-prone and very resource intensive.

Contributions

The process of designing high-confidence embedded systems focuses on the creation of digital models in order to understand and validate system properties. In order to be useful, the models should possess properties only negligibly different from its real-world counterpart while requiring fewer resources (e.g. money, time, labor) to synthesize or understand [10].

The contributions described in this thesis comprise methods to model, simulate and analyze time-triggered high-confidence embedded systems. The contributions presented in this thesis are as follows:

- **Formal Time-Triggered Model of Computation.** I have created a flexible, formal definition for time-triggered systems. The analytic properties of the model allow designers to understand the determinism, schedulability, connectivity and other properties of the design. The formal model is flexible enough to capture the design intent and execution characteristics of a wide range of the communication technologies used to connect distribute systems. From a formal system definition, it is straightforward to both synthesize system simulations and to generate executable code for realizing the actual system.

- **ESMoL Modeling Language and FRODO Time-Triggered Virtual Machine.** I have significantly contributed to the development of the ESMoL modeling language and its associated toolchain, which is used for the design, analysis, simulation, and deployment of time-triggered high-confidence embedded systems. Designers import software components defined in external modeling tools, such as Simulink, into an ESMoL model. Details about the hardware platform are joined with the component definitions into a full description of the embedded system. Analysis tools for time-triggered schedulability and controller stability have been integrated and can be applied against an ESMoL model. C-based functional code can be synthesized directly from the system model. I have developed the FRODO virtual machine (VM) which is the light-weight runtime layer that implements time-triggered execution and communication semantics. It is upon the FRODO VM that functional code generated from an ESMoL model executes.

- **Automated Synthesis of Time-Triggered TrueTime Models.** I have developed an extension to the ESMoL toolchain that automatically synthesizes a Simulink model for the analysis of platform effects. The TrueTime toolbox [11, 12] for Matlab/Simulink is a set of reusable blocks that facilitates the development of models that include the execution behavior of both hardware and software components. Building on top of TrueTime, I developed a version of the FRODO runtime that provides a time-triggered execution environment. Functional code is generated from an ESMoL system model and integrates with the TrueTime runtime code. Additionally, a new Simulink model, with the appropriate TrueTime blocks for the given hardware configuration, is automatically synthesized. This model combined with the generated code is capable of simulating possible platform effects introduced by the deployment of the time-invariant controller model onto the hardware platform.

- **Integration of the ESMoL and BIP Languages and Toolchains.** I have created a translation for models defined in the ESMoL language to the BIP language [13, 14]. The BIP language and toolchain are similar to ESMoL in their ability to model software components and systems. The BIP language was expressly designed with analysis of models in mind, though the toolchain also supports various methods for simulation and execution. The BIP language is able to express and compose heterogeneous models of computation, and automatically simulate them on a distributed cluster of machines. My integration of the BIP and ESMoL toolchains furthers the goal of virtual prototyping for ESMoL-defined systems through use of BIP's analysis tools and its heterogeneous and distributed runtime engine.

These contributions provide a robust theoretical and practical foundation on which to design, analyze, synthesize, and deploy time-triggered systems into safety-critical applications.

Thesis Organization

Chapter II provides a detailed survey of the background topics directly related to the research for virtual prototyping of high-confidence embedded systems. This includes time-triggered architectures, platform effects modeling, and heterogeneous distributed simulation approaches. In Chapter

III I present the formal definition for time-triggered systems and explore several analytic properties of this model. Chapter IV discusses the ESMoL language, its toolset, and the FRODO virtual machine for the realization of designs of time-triggered systems. In Chapter V I present the automated synthesis of TrueTime models for platform effects analysis from an ESMoL system model. In Chapter VI I present the integration of the ESMoL and BIP modeling languages and tool chains for distributed heterogeneous simulation. In Chapter VII, I conclude this thesis. Additionally, a detailed bibliography for all referenced literature is included.

CHAPTER II

RELATED WORK

The complex interplay of software, hardware and physical components in cyber-physical systems necessitates simulation and analysis of the system in order to gain a better understanding of its behavior [15]. Theoretical analysis and model-based simulation also help to reduce the cost and risk of developing new systems as compared to building physical prototypes [4]. A toolset for rapidly building detailed models of embedded systems, which can be analyzed and simulated, would be very useful for designers, but such a toolset must be able to address certain issues.

Safety critical, or high-confidence, distributed embedded systems add additional complexity to the design process. Their behavior must be very well understood both functionally and temporally. Strong architectural constraints, such as being strictly time-triggered, are typically placed on such systems in order to make analysis of their behavior simpler or, in some cases, even possible. High-confidence embedded systems exacerbate all of the issues found in typical CPS designs because of the need to ensure system safety, reliability, and robustness. Simulations of high-confidence systems must be more detailed and accurate, and must take into account the additional behavioral constraints placed on these systems.

Creating a toolchain for building detailed models of distributed high-confidence CPS systems requires leveraging multiple existing domains. This thesis directly builds upon the previous development efforts of the ESMoL language and toolchain [16, 17, 18]. The remainder of this chapter reviews the existing literature and background information about the three challenges outlined in the previous chapter. First, existing time-triggered formalisms, platform architectures and alternatives to the TT approach are discussed. Second, current methods for analyzing the impact of platform effects are covered. Finally, existing theory and techniques for creating large-scale, distributed and heterogenous simulations is reviewed. These topics form the foundation for the research presented in this thesis and directly inform the approach and methods undertaken.

Time-Triggered Systems

In this section I provide a detailed review of the time-triggered (TT) approach to system design, various time-triggered communications networks, and selected alternative protocols.

The Time-Triggered Approach

In distributed safety-critical systems, reliable and predictable communication must take place between nodes. The communications medium connecting the nodes must maintain desired properties, such as timeliness or fault tolerance. Ideally, these properties are validated analytically on individual system components and then validated components are composed together while maintaining properties. Such a constructive approach makes building larger systems far simpler compared to revalidating an entire system after each component is added.

There are two primary forms of communication networks: event-triggered and time-triggered [4]. In an event-triggered network the occurrence of an event on a node triggers the transmission of a message. The temporal control, or timing, of an event is dictated solely by the internal state of the sending node and without regard to the state of the network or other nodes. Validation of properties for this one node, in regard to external communications, could be straightforward. However, composition of event-triggered nodes is not possible because the temporal control of the communications network does not lie at a global level, but is shared internally by all nodes. Thus, adding an additional node impacts every other node.

Event-triggered protocols employ a variety of techniques to coordinate among nodes. CAN [19, 20] uses a priority-based arbitration to determine which node will send. Ethernet uses a simple random access scheme. Other approaches uses tokens. Regardless of the approach, the individual node still determines its temporal access to the network, and therefore denies the use of composition.

In time-triggered communications a global schedule dictates all network transmissions. The schedule determines when all messages are sent and received and any deviation from the schedule may be treated as an error. The global schedule also allows for straightforward addition of new components. The designer must simply find times in the global schedule suitable for both the component and the existing schedule. The time-triggered approach allows for composition, and thus

eases the design burden for distributed real-time systems.

Conformance to a time-triggered schedule imbues the communications channel with robust fault detection properties. Because all transmissions should occur according to the schedule, any deviation, either failure to transmit or transmission at the incorrect time, are easily detected. This stands in contrast to event-triggered networks where the arrival of a message is the only indication that one has been sent and a failure to send cannot easily be detected beyond the sending node. In a time-triggered system, failure of a node is detected by other nodes immediately upon a failure to receive a scheduled message. Additional fault-tolerance mechanisms, such as CRC for message corruption, are compatible with time-triggered messaging and can be integrated without compromising fundamentals of the approach.

The time-triggered approach is not without compromise. The global schedule is created statically based upon the configuration of the system. This works well as long as the system configuration is static throughout execution. Dynamically evolving systems are becoming more common and adaptation of time-triggered principles is not straightforward. For example, the flight-control system of an unmanned aerial vehicle (UAV) makes an ideal candidate for a time-triggered approach. Its components are well defined, its dynamics are understood, and its configuration is static throughout its nominal flight envelope. However, a "flock" of network controlled UAVs, flying in coordinated flight begins to stretch the applicability of the time-triggered model of computation. UAVs can join or leave the flock at will. Network communications between UAVs will be wireless and subject to unbounded delays and losses. Time-synchronization would be poor. Compositionality in such a system would be an ideal property as UAVs could easily be added or removed while maintaining flock safety, but strictly time-triggered architectures are not yet capable of satisfying such demands. Research on loosening the TT approach's constraints [21, 22] may yield benefits in this area.

An additional compromise made by time-triggered execution is a trade off between bandwidth utilization and sporadic events. Take the case of a catastrophic component failure. This failure is highly unlikely to occur but when it does the entire system should be notified immediately. In a strictly time-triggered architecture the transmission of this message must occur according to the global schedule. Therefore the global schedule must allocate time for its transmission every

hyperperiod even though the probability of using that transmission slot is essentially zero. What should be done if numerous systems have the possibility of sending such high-priority, but low-probability messages? Some combination of time-triggered and event-triggered would seem to be needed.

Time-Triggered Protocols and Platforms

There are a number of implementations and interpretations of a time-triggered architecture. Early work before the codification of time-triggered principles, such as the MARS project [23], focused simply on providing robust distributed communications. A key concept embraced by the MARS project is called "fail-silent," which means that a node either sends the correct message or no message at all, thus avoiding byzantine failures. Access to the communications bus is through a simple TDMA scheme with a static schedule. Other aspects of future TT implementations (i.e. time synchronization, statically computed task execution schedule, self-checking features) were already present in this system. The principles developed from this project served as the basis for development of numerous time-triggered communications protocols.

The Time-Triggered Protocol & TTP/C

Further development of the theory and practice of time-triggered architectures led to the development of the Time-Triggered Protocol (TTP) [24], later generalized into the Time-Triggered Architecture (TTA) [5], and the TTP/C implementation [25]. TTP/C is an end-to-end communication system consisting of both custom hardware and driver-like software components installed on each node. A message is transmitted via a TTP frame. Each frame consists of: start of frame, control field, data, CRC, and inter-frame delimiter fields. The control field contains the identifier for the sender, the sender's understanding of the current global time, and the sender's membership knowledge, discussed shortly. The data field is next and contains the actual message being transmitted. Finally, the CRC is a 16-bit code capable of detecting all single bit errors.

Additional services are layered on top of the core message-passing protocol. Membership services provide a means to track the state (valid or not) of other nodes. Each node must maintain a digest

of which other nodes in the ensemble are in a valid state. Each time a node sends a message it concatenates its membership knowledge into the control field of the TTP message. By receiving the sent message at the appropriate time, other nodes consider the sender, as well as its membership information, valid. By sharing membership knowledge in this manner, it is possible to quickly identify the state of nodes as well as the state of communication links. For example, if node A believes node B to be in an invalid state because of an undelivered message, but it receives membership information from node C that it believes node B to be valid, then it is probable that the network link between A and B is compromised.

Two other important services are time synchronization and cluster initialization. As stated previously, the local clocks on every node must maintain tight time synchronization throughout execution. This requires an initialization protocol as well as ongoing synchronization. Initialization takes place after each node has started up, does not require a designated "master", and is fault tolerant. Called "cluster startup", each node first listens for a startup message from any other nodes. If one is received, the initial clock value is set and ongoing synchronization begins. If no startup message is received, the node sends out its own startup message. Ongoing synchronization takes place using normal message-passing. The control field in a TTP message contains a clock value. Additionally, each message transmission takes place according to the global schedule. When a message is received, an offset is calculated from the difference between the actual receipt and the scheduled receipt. Over the course of a hyperperiod these offsets are recorded. TTP/C adopts the averaging technique presented in [26]. The technique is able to deal with f faults by discarding the f highest and f lowest offset values and applying some averaging function to the remainder. This technique is proven to converge over time across a set of nodes with bounded local clock drift and bounded communication delays. Using this technique, TTP provides robust ongoing clock synchronization as long as messages are being transmitted.

TTP/C does support limited asynchronous message passing. Some portion of each message can be reserved for event information. Obviously, there are strong restrictions on this approach. First, asynchronous message bandwidth cannot be shared among nodes. Second, events are only communicated according to the static schedule and must wait until the next message passing round.

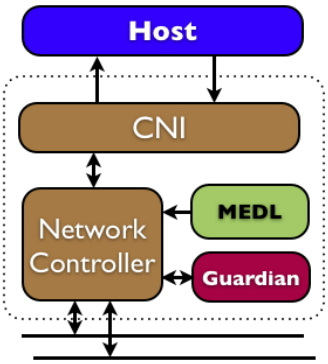

Figure 2: TTP/C Node Architecture.

The TTP/C architecture consists of four primary components: the host or node, the communications network interface (CNI), the network controller, and the bus itself, as shown in Figure 2. The host is any computing device that wants access to the bus. The CNI is the interface between the host and the rest of the network. It acts as a **temporal firewall**, isolating the host from the network and not allowing any control errors to propagate. The TTP/C network controller implements the time-triggered protocol and regulates access to the physical bus. It is within the network controller that the message descriptor list (MEDL) resides. It is the MEDL that contains the global static message transmission schedule. The bus physical layer is not specified. However, the specification requires that the physical layer must provide two independent channels, be a broadcast medium, and propagation delay must be bounded. The CNI and controller are available in both software and hardware implementations. The software is more flexible and configurable, while the hardware is more suitable for production environments and is more fault tolerant.

The suite of tools supporting TTP/C development has grown over time. Initially they lacked a comprehensive modeling environment and other tools necessary for development. Now TTTech, the company behind TTP/C, provides a strong suite of proprietary tools that support the end-to-end development cycle, from integrating with Simulink through DO-178B certification. Perhaps the strongest arguments with TTP/C are its proprietary nature, its use of custom hardware, its lack of true asynchronous message support, and its relatively low bandwidth (25 Mbps).

Efforts have been made to formally verify the TTP protocol. The clock synchronization, transmission window timing and membership services within TTP have all been formally verified in isolation [27, 28]. However, their emergent properties when combined are more complex and are not fully answered question.

Time-Triggered Ethernet

The time-triggered ethernet, or TTE, standard [29] adapts the TTP to ethernet-based networks and expands the protocol to support both time-triggered safety-critical traffic as well as other, lower-priority traffic. Initially, research was done to directly adapt TTP/C to use a COTS ethernet infrastructure, but the resulting throughput was unremarkable [30]. Instead a new adaptation of the time-triggered protocol was needed to reflect the very different nature of full-duplex switched ethernet. The new protocol seeks to add determinism to ethernet in order to provide both robustness and certifiability. Figure 3, adapted from [29] Figure 4, illustrates a typical safety-critical TTE network configuration.

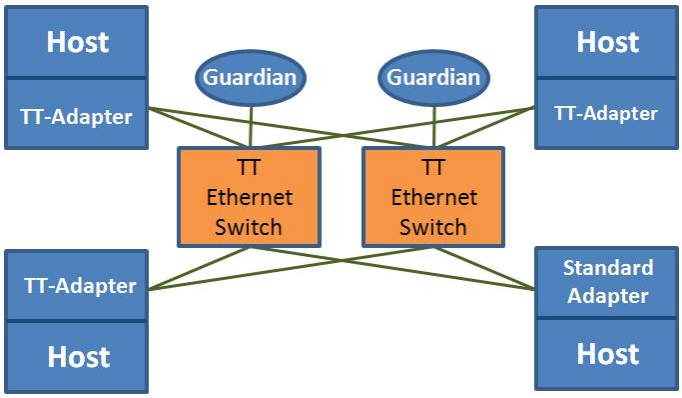

Figure 3: Example Time-Triggered Ethernet (TTE) Network for a Safety-critical Application.

The logic for handling and routing traffic is contained within a specialized ethernet switch, seen at the center of the figure. Standard ethernet host adapters, protocol stacks and applications are able to be integrated into the network without alteration. The essence of the scheme is that the centralized switch prioritizes all time-triggered traffic over non-time-triggered messages. TTE uses the standard ethernet header to identify a message as being time-triggered or not. Internal to a

TT message is another header that defines the type of TT message. The highest criticality message type is protected time-triggered message (PTTM) and corresponds to a TTP/C message. PTTM messages are transmitted according to a static schedule and are guaranteed a known transmission delay and minimal jitter. The TTE switch preempts all other traffic for PTTM messages. In Figure 3 a guardian is attached to each TT switch. The guardian monitors PTTM messages and can disable ports on the switch if a faulty host is detected. The next highest priority is for unprotected time-triggered messages (UTTM). These are identical to PTTM but do not have the additional protection of a guardian, which is necessary for certification. Below PTTM are free-form time-triggered messages (FFTT). These are simple schedule-driven messages that have priority over event messages. TTE also uses startup and time synchronization messages to initiate and maintain time synchronization between hosts in a cluster. The lowest priority messages are event-triggered messages (ETM). They can be sent by any node at any time but will be preempted at either the node or switch level by any other traffic.

TT switches and host adapters are custom built hardware designed to support the TTE protocol. Software implementations of these components are possible but do not provide fault tolerance certifiability. Hosts that send only event-based messages are able to use commodity software and hardware to connect to the network. This reliance upon proprietary hardware and software are again a point of issue. The TTE tool suite leverage many aspects of the TTP/C suite but are not yet as mature.

SAFEbus

The SAFEbus [31] architecture, standardized as ARINC 659 [32], was developed by Honeywell for the Boeing 777. The SAFEbus standard specifies dual redundant bus interface units (BIU) per node, with the bus wiring itself consisting of quad-interconnects, again paired into two separate busses for redundancy. The dual BIUs monitor each other and provide self-checking such that if either BIU thinks the other should not be transmitting, neither BIU can transmit. Incoming messages sent on the dual busses are compared and only bit-exact messages are passed to the host application. Access to the bus is granted through a static TDMA schedule stored in a table in EEPROM hardware. The

architecture for SAFEbus nodes is shown in Figure 4.

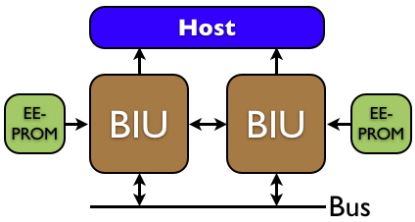

Figure 4: SAFEbus Node Architecture.

The access scheme is time-triggered, though the creation of this bus predates the codification of time-triggered principles in the MARS project. Higher-level services, such as time synchronization and membership tracking, are provided transparently to host applications. Overall, multiple levels of redundancy are built into all aspects of the architecture. While this provides high levels of reliability and fault-tolerance, the result is very expensive and proprietary hardware. As a result, it has not seen much adoption beyond the 777 aircraft, but has a strong safety record thus far.

Giotto & TDL

The Giotto [33, 34] language and associated tools are based on time-triggered execution. Giotto extends the semantics of the TT approach to include the time-triggered invocation of tasks, mode switching and message passing. A designer models a system using a textual program that defines a set of tasks, modes, and mode switches. A mode defines a set of tasks and and a set of mode switches that move between modes. A program is always in one, and only one, mode at any given time. Once a fully platform-independent model is defined, compilation can be attempted. The Giotto compiler attempts to map the program onto a hardware platform, perform scheduling, and assign tasks to nodes. A fully automated mapping is not always feasible and the designer may be prompted to add manual annotations. User driven annotations can help reduce the complexity of mapping platform-independent models onto actual hardware platforms. Annotations fall into one of three categories: hardware platform, task mapping and scheduling annotations.

In [35] the Giotto authors introduce fixed *logical execution time* (LET) semantics. LET assumes that execution times associated with tasks are independent of the actual amount of time it takes the

task to execute on hardware. Additionally, in LET all inputs to the task are assumed to be present at the beginning of its execution time and all outputs are given at the end of its execution time. The LET assumption completely separates the logical temporal design of a system from the actual execution time consumed on hardware. Figure 5, adapted from [36] Figure 1, illustrates this concept.

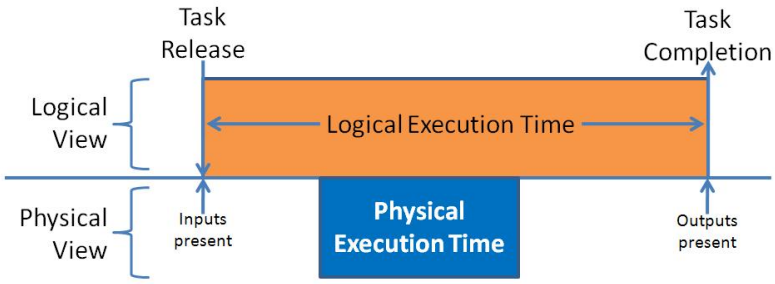

Figure 5: Logical Execution Time (LET) Separates Logical Temporal Design from Actual Execution.

In some ways Giotto represents a reversal of focus from TTP/C and TTE, in that while these two protocols focused entirely on the communications network, Giotto makes broad assumptions that the network will provide TT behavior, and instead focuses almost entirely on system modeling and task execution. In fact, [33] states that Giotto could run atop either existing TT network. In their literature, Giotto's authors make use of wireless (infrared and 802.11) communications networks. It is not explained how robust time-triggered behavior was achieved on such error-prone networks.

The ongoing development of Giotto led to the creation of TDL [37, 36, 38]. TDL extends Giotto with a few additional notions but does not stray from logical execution time and time-triggered semantics. The most notable addition to TDL is the use of modules. Modules are analogous to components as they support local definition of variables, constants, tasks, modes, and inputs and outputs. Modules also allow for globally asynchronous, locally synchronous behavior to be added. Messages between modules may be passed asynchronously while all messages internal to a module must be synchronous. Finally, TDL further abstracts the design of a system from its physical hardware with the concept of *transparent distribution*, where all notions of a hardware platform are separated from the system definition. TDL tools have evolved from being a purely academic pursuit to being commercially available [39].

FlexRay

A newer entrant into the automotive networks domain is FlexRay[40]. FlexRay is described as a combination of BMW's Byteflight [41] protocol and TTP/C, and seeks to be suitable for X-by-wire applications [42]. The FlexRay communications cycle, or hyperperiod, is divided into four primary segments: static, dynamic, symbol window, and network idle time, shown in Figure 6.

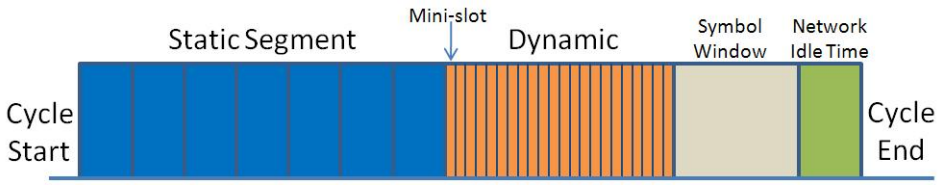

Figure 6: A FlexRay cycle broken into its four segments: static, dynamic, symbol window, and network idle time.

FlexRay provides both time-triggered and event-triggered execution models. It accomplishes this by grouping types of events into one of the cycle segments. Time-triggered communications are scheduled individually within the static segment. A major difference between FlexRay and TTP/C is the way in which event-triggered communications are handled. As mentioned above, TTP/C allows designers to reserve space in TT messages for event information. In FlexRay, event traffic must be communicated within the dynamic segment of the cycle. The dynamic segment is divided into "mini-slots", each a *macrotick* (the smallest practical unit of time in FlexRay) long. Each mini-slot is assigned to a particular node, the higher the priority of the node the closer to the beginning of the dynamic segment. If a node needs to send an event-based message it waits until its mini-slot in the dynamic segment, checks to see if anyone else is sending, and if not, it begins sending. The duration of a dynamic transmission may only last until the end of the dynamic segment. With this approach, time-triggered traffic maintains its determinism but event-triggered messages can also be sent. This scheme does include the possibility of one node with an early mini-slot always sending and thus starving the remaining nodes. The symbol window segment is reserved for maintenance and identification of special cycles, such as startup and initialization. The network idle time segment is a reserved block of time used to allow nodes to adjust this clocks as part of the clock synchronization

scheme.

The FlexRay protocol incorporates many of the other services seen in TTP/C, such as fault tolerant clock synchronization, CRC error checking, and bus guardians. It does not include any form of membership services, its bus guardian is not fully independent from the communications controller, and it also lacks several other points of redundancy and fault tolerance found in TTP/C [42]. As a result, its adoption within more safety-critical applications, such as avionics, is unlikely. While FlexRay seems to have achieved critical mass within the automotive industry, in late 2009 the governing consortia disbanded, leaving its ongoing development in question.

Other Implementations

Several other implementations of time-triggered architectures exist. In [18] a simple time-triggered virtual machine (VM) was developed that executed using a standard I^2C bus and commodity nodes. This VM was integrated into an early form of the ESMoL tool suite discussed in Chapter IV of this thesis.

Other efforts involve mapping domains onto time-triggered behavior. For example, in [43] the authors migrate a controller model from Simulink, through SCADE\Lustre [44], and finally onto TTP/C. SCADE is an environment for developing safety-critical applications built upon the Lustre language. Synchronous languages map well to time-triggered execution as long as feasible schedules are able to be found. The DECOS project [45, 46] builds further on SCADE and is directly integrated with TTE. These mapping approaches generally do not extend or enhance the semantics of TT behavior but leverage the architecture as a robust platform for execution.

Alternatives to Time-Triggered Execution

The time-triggered architecture is not the only approach used in safety-critical systems. Taking avionics as an example, FAA regulations only dictate the use of a space and time partitioning kernel [47, 48, 49], specifically the use of an ARINC 653 [50] compliant kernel. The use of a partitioning kernel does not preclude the use of time-triggered scheduling, but the standards do not mandate

the use of TT protocols either. The automotive industry is also moving in the direction of time-triggered approaches, but their standards do not yet mandate the use of partitioning kernels nor time-triggered communications.

Because partitioning kernels, by definition, partition temporal access to resources, it is possible to build time-triggered execution on top of a partitioning kernel. With this approach, scheduling within the kernel and across communication networks becomes the key factor. Time-triggered execution within a node can be achieved on an ARINC 653 kernel by having a single, non-preemptable task inside each partition, and the task's period matches the partition's. ARINC 653 schedule analysis tools must support this approach. Currently, major scheduling analysis tools, such as Cheddar [51], do not explicitly support the creation of time-triggered schedules for partitioning kernels. The ESMoL schedule analysis tool [52] is being extended in conjunction with the MBSHM project [53] to support time-triggered scheduling for ARINC 653 kernels. Alternatively, the POK kernel [54] offers an ARINC 653 compliant kernel and integrates with the Ocarina [55] AADL-based [56] tools. AADL does not explicitly support time-triggered execution for either tasks or communications, but its language primitives are flexible enough to capture time-triggered semantics. Once a system model has been defined in AADL, the Ocarina tool translates the model into ARINC 653 compliant interface code which is compiled into a POK-based executable [57].

There are numerous alternatives to time-triggered communication networks for safety-critical systems [58]. For all non time-triggered protocols there is some form of contention resolution algorithm. Inevitably, these algorithms lead to some form of non-determinism in the network's behavior. Additionally, almost every one of these networking protocols is accompanied by some proprietary hardware standard. Current trends are beginning to sway towards the use of standardized hardware, but this evolution in approach will take time in order to calm doubts about the reliability of such systems. Below we discuss several existing and emerging communications networks that attempt to provide safety-critical reliability.

ARINC 429

The majority of civil aircraft designed in the last thirty years use the ARINC 429 standard [59] for inter-line replaceable unit (LRU) communication. This standard uses redundant twisted pair wiring and redundant transmit and receive interfaces. Communications is based on a 32-bit word at speeds of either 12.5kbps or 100kbps. ARINC 429 avoids transmitter contention, and therefore non-determinism, by allowing only one transmitter per wire pair. In other words, all connections between LRUs are point-to-point requiring large amounts of wiring in complex systems.

ARINC 664 - AFDX

The three most recent large commercial aircraft, the Boeing 787 and the Airbus A350 and A380, all have adopted the Avionics Full-Duplex Switched ethernet (AFDX), ARINC 664 [60], standard. The standard is built on top of the 100 Mbps 802.3, IP, and UDP standards with additional protocols layered above UDP. Therefore, AFDX can leverage "standardized" ethernet hardware for both switches and host (called *end-systems* in AFDX) adapters. Software only AFDX implementations are available [61]. Actual deployments use specialized hardware both to increase reliability and fault tolerance, but the underlying ethernet standard is unchanged, so in theory non-specialized hardware and software could be integrated. The core of the AFDX specification is the definition of *virtual links* (VL). Each VL contains exactly one transmitter and one or more receivers. The VL is given a *bandwidth allocation gap* (BAG), in milliseconds, which is the minimum interval between messages, and an L_{max}, in bytes, that is the largest ethernet frame that can be sent. Therefore, maximum bandwidth for a VL with can be expressed as:

$$\text{Maximum Bandwidth} = L_{max} \times 8 \times (1000 \ / \ \text{BAG}) \tag{1}$$

For example, a VL with L_{max} of 200 bytes and a BAG of 32ms yields a maximum bandwidth of 50,000 bits per second. The AFDX specification also dictates bounds on the maximum jitter allowed for any end-system. This non-deterministic behavior arises from the fact that the senders on multiple VLs are free to transmit at the same time to the same receiving end-system. In this

case the ethernet switch buffers the messages in some order and then transmits them sequentially to the receiver, thus incurring some variable delay. Jitter is a function of the link bandwidth, Nbw, typically fixed at 100Mbs, and the L_{max} of all VLs on that link:

$$\text{Maximum Jitter} \leq 40\mu s + \frac{\sum_{j \in \{setofVLs\}} ((20 + L_{max_j}) \times 8)}{Nbw} \tag{2}$$

Under no circumstances is the maximum jitter allowed to exceed 500μs. So, while the AFDX standard does allow non-deterministic behavior is does provide strong bounds on the maximum jitter. Internal to each end-system is a scheduler that enforces both the L_{max} and BAG for all outbound messages. Redundancy for this can be implemented by adding functionality to the central ethernet switch to additionally enforce these values for each end-system. This additional redundancy comes at the cost of AFDX specific ethernet switches.

Redundancy in an AFDX network is provided by mandating every end-system be connected to an A and a B network. A very simple sequence number scheme, 1-byte with rollover, checks to see if any frames have been missed. Frames with identical sequence numbers arriving from the two networks are compared and only matches are forwarded on to the host application. Based on its widespread adoption within the avionics community, AFDX clearly provides enough robustness and certifiability for safety-critical applications despite its non-deterministic behavior.

CANbus

Safety-critical network standards have also evolved from within the automotive domain. Compared to avionics standards, these standards tend to focus less on robustness and more on flexibility and cost. The dominant automotive network protocol is the controller area network (CAN) [19, 20]. A CAN network consists of two-wire interconnected to each node with a 1Mbps bandwidth maximum, less over longer distances. Every node can read and set the voltage on each of the wires. CAN is an event-triggered bus. Contention for the bus is handled using Carrier Sense Multiple Access/Collision Detection with Arbitration on Message Priority (CSMA/CD+AMP). The content of each message in the system is labeled with a globally unique identifier. In part, the selection of an identifier is driven by the priority of that message. If two nodes try to send at the same time they sense that

a collision has occurred. The node sending the higher priority message is given access to the bus. The unique identifier is also used by each receiving node in an acceptance test to determine if that node will process the message further.

One approach to reducing the likelihood of contention is to reduce the expected utilization of the bus. The lower the utilization the lower the chance of a collision. One study, [62], found most CAN implementations had a designed utilization of roughly 50% for non-critical functions. For safety-critical functions typical utilization was 10-20%.

Some work has been done to bring time-triggered execution to the CAN bus, called TTCAN [63]. In TTCAN, a global schedule is created and all nodes execute according to the schedule. Global time is maintained by the frequent broadcast of a "Reference Message" meant to signal the start of the next *Basic Cycle* (BC). In each BC a number of messages are exchanged based on the schedule. Each BC ends with the broadcast of the next reference message. After some design-dependent number of BCs, the entire schedule, called a *Matrix Cycle*, repeats. With this approach, no bus contention should occur and TDMA-style access will proceed. Wire level signaling is identical between CAN and TTCAN allowing for the reuse of some circuitry designs. TTCAN appears to remain an academic research project at this time.

Platform Effects Simulation

In this section we discuss methods for understanding the impact introduction of a distributed hardware platform will have on the performance of a software-based controller. Impacts can arise from any number of root causes, including, but not limited to:

- **Temporal and scheduling**: Deployment of a time-invariant controller onto a platform that realizes an alternate model of computation can significantly impact the controller's temporal performance. For example, the imposition of a time-triggered execution and communication schedule introduces non-trivial delays between sensing and actuating, which can materially impact controller performance.

- **Numerical differences**: During controller design the numerical resolution of the underlying hardware platform may vary significantly from the controller design environment. For example,

a given microprocessor may only support 16-bit floating-point math, or may even only support fixed-point math, while Simulink makes a default assumption of 32-bit float data types.

- **Platform flaws**: In an ideal world, hardware platforms are flawless implementations. In reality, hardware platforms contain known and unknown flaws. Little can be done during the design phase to account for unknown platform flaws, but known platform issues can be accounted for.

- **Error conditions**: During nominal execution, a deployed controller's performance may closely align with its idealized performance, but under fault conditions the hardware platform can play a much larger role. Loss of a node, loss of a network link, corrupted data, etc.; all fault hypotheses must be understood and mitigated.

Hardware/Software Co-Simulation

Co-simulation [64] involves many aspects of system-level performance simulation. A key aspect of co-simulation is the use of a high-fidelity model of the hardware platform. Simulation of the involved software components takes place within the hardware simulation, i.e. the hardware simulation is a virtual machine within which the software executes. As long as the hardware model is of high enough fidelity, co-simulation techniques are able to achieve very accurate results.

The hardware platform model in co-simulation is represented at very a low-level of abstraction. Hardware models may be specified as low as at the RTL level using languages such as VHDL or Verilog [65]. With the platform specified at this level of detail, nano-second resolution is possible [66], but computationally expensive. Higher-level system abstractions, such as timed automata or time-triggered execution, could be incorporated into the software component models but are typically not.

Co-simulation models tend towards such high levels of detail that they are good predictors of system performance, but at the cost of requiring extremely detailed hardware models and very slow simulation execution times. The majority of co-simulation research has focused on single node, or system-on-a-chip (SoC), simulation [67]. Modeling and simulation of large-scale distributed systems using co-simulation is not yet common, perhaps due to its slow simulation time or the requirement

for detailed hardware models.

System-Level Performance Simulation

When idealized, or time-invariant, controllers are deployed onto real hardware platforms their performance may be altered. Deployment onto a distributed architecture can impact performance even more. Identifying these properties, understanding their impact, and mitigating any negative impacts early in the design process, are very important considerations. Static analysis techniques are able to determine some of these properties, but others can only be determined via simulation. Numerous methods have been developed to simulate software components running in conjunction with hardware. By its nature, simulation does not provide a comprehensive analysis of a system, but instead provides execution traces which help to understand a system's properties.

Current state-of-the-art model-based controller development environments, such as Simulink [68], do not directly support the concept of a deployment platform and do not natively simulate the impact of deployment on controller performance. Third-party extensions to Simulink have been developed that allow these impacts to be simulated and analyzed. The TrueTime toolbox [11, 69] is a suite of Simulink blocks designed expressly for this purpose. TrueTime supports modeling, simulation, and analysis of distributed real-time control systems including real-time task scheduling and execution, various types of communications networks, and "analog" inputs and outputs for interaction with the continuous-time plant model. While gaining insight into platform effects is crucial, TrueTime imposes an additional burden on systems engineers. It requires significant effort and a deep understanding of both TrueTime and the desired deployment platform in order to adapt time-invariant models into TrueTime models.

TrueTime's flexibility allows for it to model a wide range of real-time platforms, from simple systems through complex hard real-time architectures. There are numerous examples of TrueTime being used to simulate platform effects [70, 71]. In all of these examples the models are manually created and details of the hardware platforms are translated into TrueTime blocks by hand. Additionally, no examples were found where time-triggered communication was employed to provide a more robust distributed control system architecture.

Architecture definition languages (ADL) encompass the abstract modeling of both software components and hardware platforms [72]. This approach is a holistic means to capture the system details required in order to faithfully simulate system-level behavior. UML\MARTE [73] and SystemC [74] provide a rich set of modeling primitives for specifying software components and hardware platforms for embedded systems. Both of these frameworks provide some analysis tools, though both focus more on system simulation. Neither of these frameworks explicitly supports the time-triggered model of computation for distributed systems, but both provide rich enough semantics that it would be possible to model TT execution.

Other embedded system modeling frameworks adhere to a time-triggered architecture, such as SCADE\Lustre [43], Giotto [33] and its successor the Timing Definition Language (TDL) [36], but tend not to include detailed enough modeling of the deployment hardware platform. Giotto and TDL in particular stress their "transparent deployment" approach to mapping system functionality onto the hardware platform, and thus limiting the detail of their hardware models.

Heterogeneous & Distributed Simulation

In this section, we provide a taxonomy of heterogeneous and distributed simulation approaches.

Simulation Architectures

Heterogeneous simulation is the coordinated simulation of multiple different types of models as opposed to a single monolithic model. The heterogeneous models may be of various aspects of a single system or of distinct systems within a system of systems (SoS). Distributed simulation is simply the coordinated execution of a simulation across a network of machines in order to scale the complexity or fidelity of a simulation.

Figure 7 illustrates the difference between heterogeneous distributed and monolithic simulation. In this figure the various aspects of an anti-lock braking system are simulated. Figure 7a shows how each component of the simulation is run on a separate simulation engine, depending on the engine best suited to the nature of the component. The simulation engines must then communicate to

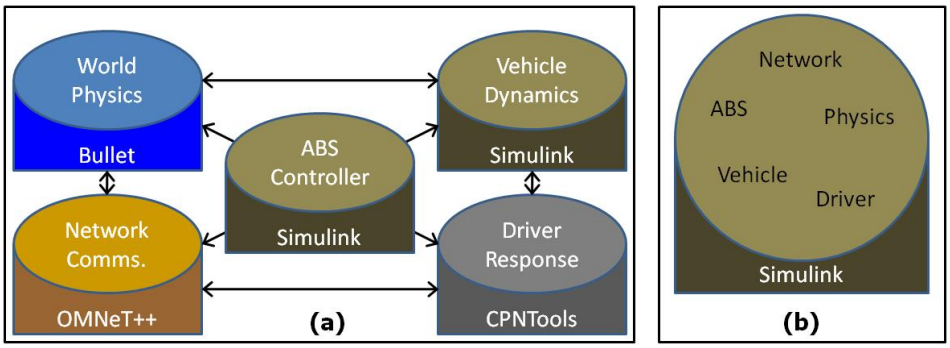

Figure 7: (a) Heterogeneous & Distributed Simulation, (b) Monolithic Simulation

coordinate the evolution of the simulation. Figure 7b depicts the monolithic approach to simulation which attempts to capture everything in a single model on a single simulation platform.

Some modeling and simulation platforms, such as Matlab/Simulink [68], Ptolemy [75], Scicos [76] and Modelica [77], attempt to offer a broad enough set of building blocks that it is practical to model complex systems entirely using one platform. While this may ease initial development efforts, for systems of sufficient complexity single modeling environments tend not to be broad enough. Additionally, it is typical for organizations to have accumulated large libraries of existing models created in diverse modeling tools which are not easily transformed into a central modeling environment. Finally, creating a single monolithic model places the full burden scaling computation on the simulation platform which may or may not support distributed execution.

Framework-based Simulation Integration

Formalized frameworks for integrating simulations can greatly reduce development workload while simultaneously allowing for more flexible and scalable simulations. The SIMNET project [78, 79] is credited with initiating research into scalable frameworks for distributed simulation. Prior research efforts focused primarily on rigid integration of a small number of stand-alone simulators. While not a truly heterogeneous platform, SIMNET was able to integrate simulations for disparate purposes into a single virtual environment.

The SIMNET domain was discrete simulation of battlefields and combat vehicles. Soldiers used SIMNET as a live training environment. Full-scale vehicle mockups were installed in various locations around the world and networked together. The controls of these vehicles and visual displays

for their occupants were directly connected to and controlled via SIMNET. The SIMNET language was a simple transaction modeling language with primitives for discrete event simulation and basic conditional logic. The SIMNET language described both the individual simulations and their network interactions.

One contribution of the SIMNET language was its use of dead reckoning, or incremental updates, as a means to provide position updates throughout the simulation. Use of incremental updates simplified network communications and allowed greater scaling of the overall simulation. SIMNET also developed the use of objects and events as a means to communicate environment changes [80]. Objects were owned by one node in the simulation and that node was responsible for communicating incremental state changes of the object. Similarly, events occurred as objects interacted, were created, or were destroyed.

The SIMNET architecture was strictly real-time, in the sense that the overall simulation always assumed human-in-the-loop and was not able to run at either slower or faster than real-time. In part this was due to the nature of the simulation. Individual nodes, and their human occupants, could enter or exit the virtual environment at any time, and each node was expected to meet specific performance guarantees about issuing object and event updates.

Due to its specific use in military simulation, the SIMNET language and framework did not gain significant acceptance outside of the U.S. military. The follow-on research project to SIMNET was the Distributed Interactive Simulation (DIS) project [81, 82, 83]. Again, DIS was developed by DARPA to support large-scale battlefield simulations. DIS built on the core SIMNET network protocols, but completely divorced the network protocols from the individual simulations. Now, individual simulations could be built using any simulation engine and integrated with the rest of the virtual environment via standardized network protocols.

As was for SIMNET, in DIS individual simulations are responsible for maintaining their own internal state as well as their understanding of the greater environment. Via the integration framework, a standardized format is used for communicating object status and all state changes. Individual simulations are able to query the state of an object via the network and are able to filter object update information as desired. Again, time coordination in DIS is negligible. Time is always assumed to be

strictly real-time and individual simulations are held to performance guarantees.

DIS protocol data units (PDU) define the data format and semantics of all interactions. The definition of PDUs is standardized by the DIS controlling body. There have been a number of updates to the collection of standard PDUs, but the context for the standardized PDUs is almost entirely military in nature (warfare, logistics, simulation management, minefield, etc.). The use of standardized PDUs closely mirrors but predates the current research into ontology-based integration methods, see Section II below. The PDUs reflect the view of an individual simulation encompassing all aspects of one whole entity, such as a tank, and not a single subsystem, such as an engine. DIS is not designed for, and not very capable of, integrating simulations of subsystems into a larger system, only of representing entities in a larger virtual environment.

Though still in use and under development today, DIS has mostly been eclipsed by the High-Level Architecture (HLA) [84, 85]. While the development of the HLA standard was again guided by the military, no aspect of the HLA framework is distinctly military. The HLA is a completely general use integration framework for simulations. An HLA simulation is composed of a *federation* of individual simulation *federates*. Shared *objects* and *interactions* are defined to which any federate may publish or subscribe. Objects are analogous to OS-style shared memory and are owned by one federate. Interactions correspond to message passing. All federation configuration information is stored in a standardized format text file called the FED file. In theory, this configuration file is portable across different HLA run-time infrastructure (RTI) implementations. The HLA standard does not proscribe a set of standardized interactions or objects. The U.S. military's OneSAF project [86], the inheritor of SIMNET and DIS's purpose, has defined a standard set of interactions and objects for their use of HLA in battlefield simulation. Other organizations have similarly published standardized federation configurations. These standardized sets of interactions and objects again correspond to the ontological approach discussed below.

The HLA standard advanced the flexibility of time control in an integrated simulation. Fundamentally, every federate must maintain a clock corresponding to the logical time internal to its simulation. This clock is distinct from any real-world "wall-time". The standard provides numerous schemes for coordinating logical clock evolution among federates. These can range from completely

lacking time synchronization, where one federate can execute arbitrarily far into the future, to completely synchronized, where all federates evolve time within a tightly bound window. Each federate can be configured to be *time-constrained* or *time-regulating*, both, or neither. A time-regulating federate's progression constrains all time-constrained federates. Likewise, a time-constrained federate's advance is controlled by all time-regulating federates. A federate that is neither constrained or regulating is free to evolve time on its own. Federates that are both evolve time in tight lockstep. If, for example, all federates can run at least as fast as real-time, and one federate tightly correlates its time advance requests to wall-clock time, then the entire federation can be made to run in real-time. Otherwise, it is possible to execute simulations both faster and slower than real-time.

Each HLA federate defines a *step size*, *lookahead* interval and *minimum time stamp*. When a federate requests to evolve its internal simulation time is does so in increments of step size, which may vary in size from step to step. Lookahead corresponds to the amount of time into the future which the federate guarantees it will not issue an interaction or object update and is generally small compared to step size. When the federate is in a Time Advance Request (TAR) state, minimum time stamp is defined as the federate's requested time plus lookahead. When the federate is in a Time Advance Granted (TAG) state, minimum time stamp is the federate's logical clock time plus lookahead. It is also important to understand that each federate maintains an understanding of all of all other federate's minimum time stamps. Figure 8, adapted from [85] Figure 5-7, illustrates how time advances happen in a federation of two time-regulating and time-constrained federates.

In this example, federate A always seeks to advance its clock in steps of size s, while federate B steps are of size 3s. Wall-clock time runs to the right, but has no units to reinforce that there is no mandatory correlation between logical and wall time. Event 1 is federate A issuing a time advance request (TAR) to the HLA run-time infrastructure (RTI) to advance its logical clock by its step size. It cannot advance its logical clock until federate B's minimum time is greater than its requested time. Event 2 is federate B issuing a TAR which immediately causes its minimum to go to T+3s since its step size is 3s. This allows federate A to change to Time Advance Granted (TAG) state and progress its logical clock to T+s. At events 4 & 5 and 6 & 7, federate A issues TAR followed

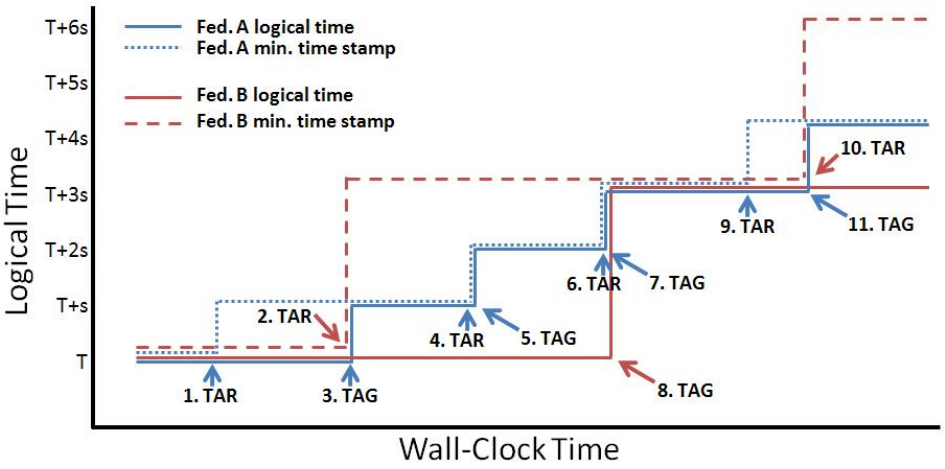

Figure 8: Example of Synchronized Time Advancement in HLA

immediately by TAG since federate B's minimum time is still greater. Finally, once event 6 has occurred, federate B can move into a TAG state and advance its logical clock. The whole sequence then begins to repeat itself.

The HLA standard still lacks numerous facilities that would be useful for developing integrated distributed heterogeneous simulations. Foremost is any formalized methodology for developing collections of interactions and objects. While a significant advantage of HLA over DIS and SIMNET is its complete divorce of the framework from the subject being simulated, the lack of a standardized lexicon of interactions and objects places a greater burden on integration designers. Another problem stems for HLA's flexibility. SIMNET and DIS simulations tended to be relatively static in the sense that the deployment configuration for a given simulation would change relatively little over time. Because HLA-based simulations are so flexible it is desirable to move a simulation from one network of computers to another for development or execution. The HLA standard does not directly address this ability and leaves such functionality up to HLA implementations or integration designers.

Each simulation model and simulation engine must be individually integrated at both the HLA API level and at overall simulation level. Simulation engine-to-HLA is simple technical integration, but simulation model-to-overall simulation integration is completely contextual, depending entirely upon its role in the greater simulated environment. Many research efforts exist relating to integrating various simulation engines via HLA, including: OPNET [87], SLX [88], JavaGPSS [89], DEVSJAVA

[90, 91] and PIOVRA [92]. As mentioned earlier, the HLA APIs provide run-time support but the problem of model integration is not addressed in these efforts.

Relevant commercial integration software does exist, such as the HLA Toolbox [93] for MATLAB federates by ForwardSim Inc. [94], MATLAB and Simulink interfaces to HLA and DIS based distributed simulation environments by MK Technologies [95]. These products focus on integration of models running on the same simulation platform and do not provide support for heterogeneous simulations.

Additionally, there have also been some efforts on enhancing HLA support by complementary simulation methodologies such as in [91] and [96]. However, these efforts, similar to those above, pursue HLA integration of isolated simulation tools. Moreover, these efforts do not have any support for model-based rapid integration of simulation tools, and limited or no support for automated deployment and execution of the resulting simulation environment.

Model-based Simulation Integration

A model-based approach holds the promise of easing the burden of developing an integrated simulation. An obvious artifact from any such effort would be a federation configuration specifying a set of interactions and objects valid for all possible constituent models. The problem then lies with understanding the domain and capabilities of each constituent simulation model and domain and determining a composition, or integration, of these that meets the goals of the overall simulation.

Multi-paradigm modeling (MPM) [97], or multi-formalism modeling [98, 99], research concerns the methods, tools, and applications related to engineering problems involving the composition of models from multiple different domains, specifically when the set of models derives from multiple modeling formalisms. Similar to MPM, the most challenging problem for distributed heterogeneous simulation is composing domain-specific models in a meaningful way.

In order to analyze complex multi-formalism systems, MPM advocates finding methods to convert each constituent formalism into some common formalism. This can be accomplished in several possible ways. First, a "super-formalism" could be found that encompasses the expressiveness of the individual formalisms, illustrated using DEVS as the super-formalism in Figure 9, from [97].

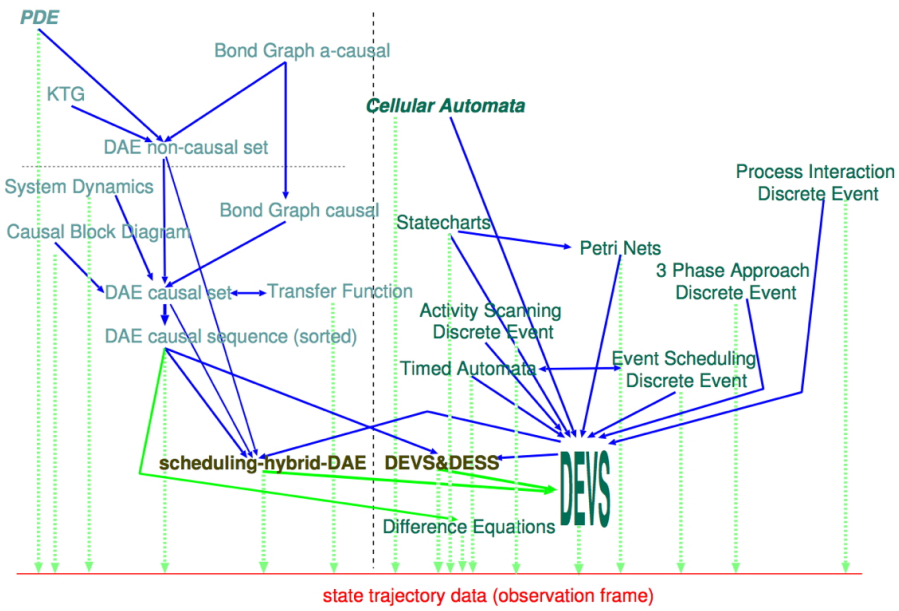

Figure 9: Multi-paradigm Modeling Using DEVS as a Common Formalism

Finding such a super-formalism is difficult, especially if the sub-formalisms are diverse in their nature. Second, each sub-model could be transformed into some common formalism. Little distinction can be drawn between this approach and the super-formalism approach, in that both require transforming models from one formalism to another while attempting to not alter the semantics of the model. As long as little or no loss of semantic understanding occurs, these approaches are feasible. But, complex systems may involve such diverse formalisms that these approaches do not work. The use of the Formalism Transformation Graph (FTG) [100] somewhat simplifies the process of understanding which transformations are feasible or not. The final approach discussed in MPM does not involve model or formalism transformation. Co-simulation is the coordinated execution of simulations of all sub-models, using their respective formalism. This approach is the basis for systems such as HLA, discussed above.

One of the other previous efforts that relates to heterogeneous simulation is the MILAN framework [101]. MILAN provides a model-based multi-domain simulation of System-On-Chip (Soc) designs and integrates several simulation tools in order to optimize performance versus power-consumption. This approach does not leverage the HLA as an integration framework, is very SoC-specific and does not attempt to be a general engine for heterogeneous simulation.

Over time communities build a common understanding by using an agreed upon vocabulary.

Specific terms and phrases take on special contextual meaning within that community. Ontology based approaches [102, 103] for simulation integration build upon this concept. A *domain* ontology is built using expertise and knowledge focused on a tightly-scoped common interest or topic, such as hydraulics, communication networks or biology. A *community of interest* ontology is built around a community with a shared set of objectives, such as military training or command and control. Finally, a *simulation tool* ontology is built to describe the functionality encapsulated within a particular simulation.

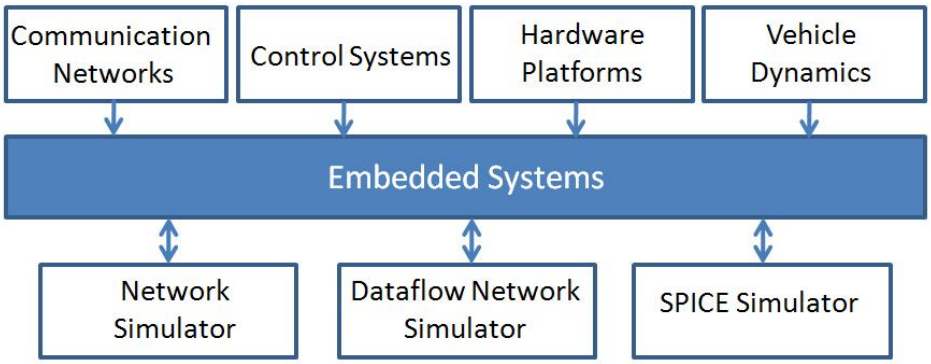

Figure 10: Example of Embedded Systems Ontology Hierarchy

Figure 10 illustrates how these three types of ontologies work together using embedded systems as the context. At the top, ontologies for specific domains are captured. These individual domains correspond to integral parts of an embedded system but are somewhat atomic in their semantics: communication networks, control systems, hardware platforms and vehicle dynamics. Individuals may be experts in these areas but are unlikely to be experts in all. At the center, a community of interest ontology incorporates the domain ontologies and is able to capture the semantics of entire embedded systems. This ontology is not necessarily a super-set of all incorporated domain ontologies as it may discard unneeded aspects from a domain. At the bottom of the figure are the simulation tool domains. Each tool may have its own formalism and semantics, in our example a network simulator, dataflow simulator and hardware simulator. The community of interest ontology must be able to map onto a set of simulation tool ontologies in order to be simulated.

A community must organize itself and come to agreement on the important terms, events and objects in order to develop a common ontology. The OneSAF HLA federation configuration mentioned

above is an excellent example of this. The military defined all of the interactions and objects of interest to them and all simulator suppliers must conform to this standard. But, the simple definition of objects and interactions does not necessarily alleviate the problem of heterogeneous integration. A deeper understanding of the semantics of these items is necessary and is not addressed in the ontology-based research. Ontologies allow everyone to speak the same words, but not always have the same understanding.

CHAPTER III

TIME-TRIGGERED SYSTEM MODEL

The idea of time-triggered execution is intuitive and easily understood at a conceptual level. While time-triggered semantics are routinely adopted in system design, a widely accepted and general abstract definition for a time-triggered system has not evolved. Individual research efforts, each with its own conceptual approach and tools, have emerged supporting time-triggered execution but they are not easily compared and tools do not interoperate.

In this chapter we present a formal model for time-triggered systems that is not tied to a specific communications technology or modeling language. From this abstract definition, numerous system properties can be derived and the behavior of the resulting system is well understood. Once defined, a model for a system can be reused across design, analysis, simulation and deployment through approaches such as model transformation and template-driven synthesis.

Definition of a Time-Triggered System

The time-triggered model of computation (MoC) is composed of tasks executing according to some statically-derived schedule. These tasks can either be computational tasks or can be communications tasks. Tasks consume messages as input and generate messages as output. The timing characteristics and grouping of tasks are what endow a time-triggered system with its properties.

We define a time-triggered system, Υ as a collection:

$$\Upsilon = \langle \Psi, \psi', M, T, \Lambda, H, Sched \rangle \tag{3}$$

In this tuple, Ψ and ψ', are a set of **Local Clocks** and a **Global Clock**. Where $\forall\ \psi_i \in \Psi$ $\exists\ \delta = |\psi_i\text{-}\psi'| : \delta \leq \epsilon$. Where ϵ is the maximum allowed divergence between a local clock and the virtual global clock for Υ. A formal definition of ϵ is given in the following sections.

- The virtual global clock value is the corrected average of the local clocks, as defined by [26].

- The virtual global clock provides a strong abstraction that allows the model of computation to not further consider the state of local clocks and instead employ the global clock in all definitions and analyses.

M, a finite set of **Messages**. Message structures are statically defined and contain a finite and fixed amount of data.

- A message type, $m_i \in M$, is analogous to a data structure definition. A message type is composed of an non-empty ordered set of typed data members (e.g. int, float, etc.).

- $m_{i,j}$ denotes instance j of message type i. Instances are the instantiation of a message type with values assigned to the data members. Message instances are used for exchange information between tasks.

T, is a finite set of **Tasks**, $2^M \times \psi' \rightarrow 2^M$. Tasks consume some finite number of input message instances and produce some finite number of output message instances. The value of the global clock may also be used as input to the task.

- The consumption and production of message instances provides a partial ordering of tasks in the system. This will be illustrated in the next subsection.

- For each $t_i \in T$ there is an associated worst-case execution time, $WCET(t_i) \in \Re^+$ and period $\pi(t_i) \in \Re^+$.

- A task has one or more instances. $t_{i,j}$ denotes instance j of task i. Each $t_{i,j}$ is given a starting time, $ST(t_{i,j})$ denoting when instance j of task i should begin execution according to the global clock, ψ'.

- Tasks may be either computational or message-passing in nature, but this characteristic has no material impact on the timing properties of the system.

- Tasks may maintain persistent internal state, but this state may not affect the timing properties of the system. The behavioral semantics of tasks and their internal state is further discussed in a section below.

Λ, is a set of **Timing Sets**, $2^T \times \Psi$, where every element, $\lambda_i \in \Lambda$, of which is composed of a set of task instances, $\{t_{a,0}, \ldots, t_{m,n}\}$, and a local clock, $\psi_i \in \Psi$.

- Against each timing set a single *timing constraint* is applied. The constraint is that all tasks within the set either execute in *parallel* or they execute in *sequence*. Parallel execution implies that $\forall\ t_{i,m}, t_{j,n} \in \lambda$, $ST(t_{i,m}) = ST(t_{j,n})$ and $|WCET(t_{i,m}) - WCET(t_{j,n})| \leq \epsilon$. Sequential execution implies that for any two consecutive tasks in λ, $t_{i,m}$ and $t_{j,n}$, that $ST(t_{i,m}) + WCET(t_{i,m}) < ST(t_{j,m})$.
- An instance of a task may be a member of more than one timing set.
- All instances of a task within a set share the same internal state.

H, the **Hyperperiod** is a repeating fixed finite interval of time, $H \in \Re^+$, during which all task instances are scheduled to execute. At the conclusion of one hyperperiod the next commences.

- From [52] the hyperperiod duration can be defined as the least common multiple of the periods of tasks in the system, $H = LCM(\pi(t_i))\ \forall\ t_i \in T$.
- Given that each task, t_i has a maximum duration, $WCET(t_i)$, and a period, $\pi(t_i)$, t_i will run $\left\lfloor \frac{H}{\pi(t_i)} \right\rfloor$ times during a hyperperiod.
- The start time for every task instance is bounded by H, $0 \leq ST(t_{i,j})$ and $ST(t_{i,j}) + WCET(t_{i,j}) < H\ \forall\ t_{i,j} \in T$.

Sched, the **Execution Schedule** of the time-triggered system. A valid schedule consists of all start times for all task instances such that the timing constraint for all timing sets is upheld. Schedulability analysis is performed offline. Schedulability analysis and its impact on freedom from deadlock are discussed in detail below.

The above abstract definition for a time-triggered system is sufficient to describe the temporal execution of a design. Further elaboration is necessary to capture additional properties beyond timing and will be discussed in later sections.

Mapping an Example System

The abstract definition of a time-triggered system given above is easily mapped to the more concrete concepts typically dealt with in designing an actual system. Computational nodes, tasks, and communication busses are present in real systems. We present a simple example of an embedded system to demonstrate mapping a real system onto the abstract definition.

Our example is a system with two nodes connected via a time-triggered bus. The precise details of the nodes and bus are not important other than that the sending and receiving nodes for a message transmission must act in unison. The beginning of each hyperperiod *Node 1* calculates a message and sends it to *Node 2*. Node 2 then processes the message, calculates a new message, and sends it back to Node 1. Node 1 then processes the message, all before the conclusion of the hyperperiod.

Formally, for this system let $M = \{m_{1,1}, m_{2,1}, m_{3,1}, m_{4,1}\}$. Let $T = \{t_{A,1}, t_{B,1}, t_{C,1}, t_{D,1}, t_{E,1}, t_{F,1}, t_{G,1}, t_{H,1}, t_{I,1}\}$. For $t_{A,1}$ the input message set is empty, $\{\}$, and the output message set is $\{m_{1,1}\}$. For tasks $t_{B,1}$, $t_{E,1}$ and $t_{G,1}$ the input message set equals $\{m_{1,1}\}$ and output equals $\{m_{2,1}\}$. For task $t_{H,1}$ the input message set is $\{m_{2,1}\}$ and output is $\{m_{3,1}\}$. Tasks $t_{C,1}$, $t_{F,1}$ and $t_{I,1}$ have an input message set of $\{m_{3,1}\}$ and output set of $\{m_{4,1}\}$. Finally, task $t_{D,1}$ takes $\{m_{4,1}\}$ as input and generates no output messages, $\{\}$. Five timing sets are defined for this system. TS_1 is a sequential timing set containing task instances $\{t_{A,1}, t_{B,1}, t_{C,1}, t_{D,1}\}$. TS_2, also sequential, contains task instances $\{t_{E,1}, t_{F,1}\}$. TS_3, sequential, contains $\{t_{G,1}, t_{H,1}, t_{H,1}\}$. Two parallel timing sets are present: TS_4 containing task instances$\{t_{B,1}, t_{E,1}, t_{G,1}\}$, and TS_5 containing instances $\{t_{C,1}, t_{F,1}, t_{I,1}\}$. Figure 11 illustrates this example system.

Figure 11 shows Node 1 containing four tasks: Task A, Task B, Task C, and Task D. Tasks A and D correspond to the initial and final computational activities described above. Tasks B is the effort consumed by Node 1 to send the message to Node 2. Similarly, Task C is the time consumed by Node 1 to receive the message from Node 2. Task H is the computational activity on Node 2. This node also contains two tasks necessary for receiving, Task G, and sending, Task I, messages. As it takes some finite amount of time for a node to either send or receive a message, this is reflected

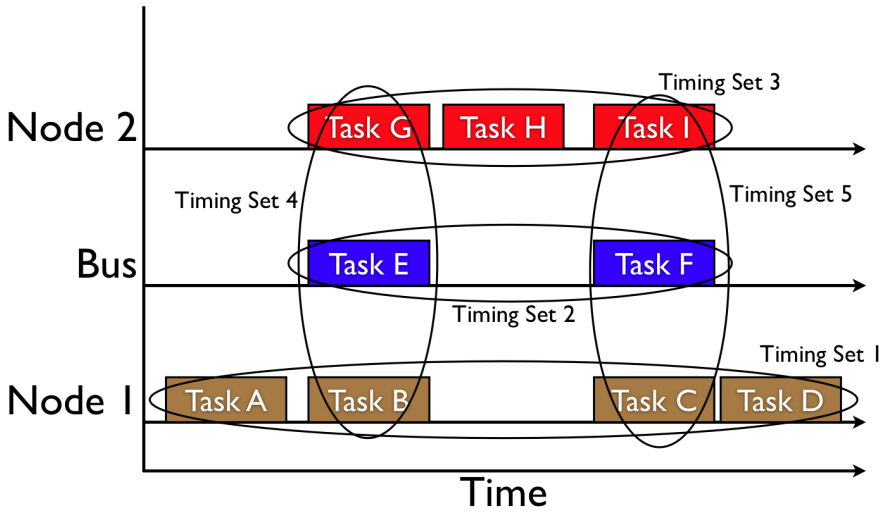

Figure 11: Example TT System with Timing Sets

by creating message-passing tasks.

Over the course of each hyperperiod our example time-triggered bus transmits two messages. Each message transmission consumes some amount of time on the bus and are represented by Task E and Task F. On our example bus, the three tasks involved in message passing (B, E, G and I, F, C) must occur simultaneously.

A simple set of rules can help guide the mapping of concrete systems to the abstract model:

1. Each computational task maps to a abstract task.

2. Physical nodes corresponds to sequential timing sets.

3. Transmission of a message involves three abstract tasks: the sender, the transmission on the medium, and the receiver. Each transmission, and its three tasks, correspond to a timing set. Whether the timing set is parallel or sequential depends on the transmission characteristics of the bus itself. This is discussed in more detail below.

4. Each time-triggered bus corresponds to a sequential timing set whose set membership consists of the transmission tasks from the messages it passes.

5. The timing set membership for physical nodes is composed of the computational tasks assigned to that node and the sending or receiving tasks for messages sent from or received to that node.

Continuing our example, using the rules above we can see that Node 1 gives us sequential *Timing Set 1* (TS_1), with membership of computational tasks A & D and message passing tasks B & C, $\{t_{A,1}, t_{B,1}, t_{C,1}, t_{D,1}\}$. Node 2 gives us sequential *Timing Set 3* (TS_3), with membership of computational task H and message passing tasks G & I, $\{t_{G,1}, t_{I,1}\}$. The bus gives us sequential *Timing Set 2* (TS_2), with message passing tasks E & F, $\{t_{E,1}, t_{F,1}\}$. Finally, the first message transmission gives us parallel *Timing Set 4* (TS_4), with the sending task B, transmission task E, and receiving task G, $\{t_{B,1}, t_{E,1}, t_{G,1}\}$. The second message transmission gives us parallel *Timing Set 5* (TS_5), with sending task I, transmission task E, and receiving task C, $\{t_{E,1}, t_{I,1}\}$.

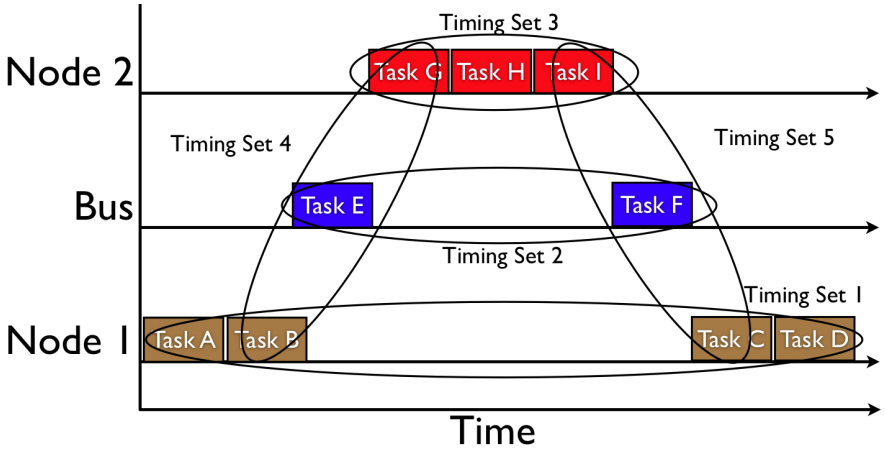

Figure 12: TT System with Sequential Transmission Timing

Different types of busses have difference timing characteristics. In the above example, the sender, the bus, and the receive all must act simultaneously. Unbuffered communications medium, such as I^2C, behave in this way. Ethernet, for example, generally works quite differently. If Node 1 used an ethernet controller to send its message it would take some small amount of time from Node 1's execution to pass the message off the its associated ethernet controller and then Node 1 could resume other tasks. The ethernet controller would then be responsible for sending the message across the bus for Node 2's ethernet controller to receive. The two controllers and the bus must act in unison, but from the perspective of Node 1 and Node 2 themselves, transmission would appear to be sequential, as shown in Figure 12.

Task Behavioral Semantics

The formal model given in the first section of this chapter defined abstract tasks and their interactions via message passing. Abstract tasks can be either computational or message-passing in nature, but for the sake of timing properties of the system the formal model makes no distinction between these. Conceptually, the time-triggered schedule defines a time in the hyperperiod at which an instance of a task is released to execute. The actual value of the input and output messages cannot effect the timing properties of the system.

Recall that our definition of tasks was $2^M \times \psi' \rightarrow 2^M$. This definition is sufficient for the temporal execution semantics of tasks, but does not completely capture their execution behavior. We extend this definition as follows: $f : 2^M \times \psi' \times S \rightarrow 2^M \times S$. S is a set of persistent variables that are shared across all instances of a task running within a timing set. f is the execution function of the task. f takes as input message instances, the global clock time, and the state variables, and outputs some number of output message instances.

Generically, tasks consume messages as input, perform some computation using the messages with some internal state and the global clock value as input, and generate some messages as output. In reality, computational and message-passing tasks are quite different in this regard. Computational tasks can assume a wide range of execution behavior while message-passing tasks simply move data from one location to another. We define the execution function for all message-passing tasks as $f(M) = M$, i.e. the identity function. The output message from the task is the set of input messages.

The execution function for computational tasks needs only to be loosely defined. The amount of time it takes to execute, regardless of input values, must be less than or equal to the stated worst-case execution time previously defined, $ExecTime(f_t) \leq WCET(t)$. Additionally, the execution function of computational tasks must be side-effect free.

The approach of implementing a task, i.e. Simulink blocks or petri-nets, is not within the scope of this thesis. Our model of computation demands only that all tasks follow LET semantics. In the following sections, we discuss several system properties that can be derived via analysis of our

formal model. None of these properties are dependent upon the implementation approach of tasks, only their timing properties. Throughout the remainder of this thesis we assume only that tasks, in their definition and execution, adhere to these requirements.

Task-Message Connectivity & Response Latency

In many designs it is paramount to understand the latency between a set of tasks receiving an input, such as from a sensor, and the generation of a response output, such as to an actuator. This duration is called the *response latency*. Using our formal model for time-triggered systems it is straightforward to calculate a system's response latency from its definition. Given a system, Υ with a set of message instances, $m_{i,j}$, tasks instances, $t_{i,j}$, with instance start times, $ST(t_{i,j})$, input and output message instances, and worst-case execution times, $WCET(t_i)$, there are two steps involved in calculating response latency.

In order to illustrate this process, we will continue with the example given in the previous section, two Nodes communicating via a TT bus. First, an execution schedule is needed for this system. One possible schedule, given a 10ms hyperperiod, is shown in Table 1.

Table 1: Example Schedule (in ms)

Task Name	Start Time	WCET
Task A	1	2
Task B	3	1
Task C	6	1
Task D	8	1
Task E	3	1
Task F	6	1
Task G	3	1
Task H	4	2
Task I	6	1

First, a directed acyclic graph (DAG) is built based on task-message connectivity as described by the system definition. Every task is said to consume some, possibly empty, set of message instances and produce some, possibly empty, set of message instances. It is simple to create a graph given a set of tasks and the sets of message instances they produce and consume. Each task instance is a node, and nodes are connected if one task produces a message instance another task consumes. Tasks in

parallel timing sets can be thought of producing and consuming the same message instances, since this generally represents a transmission, and therefore are seen in parallel in the graph. Continuing our example, Figure 13 illustrates the task-message graph for our system.

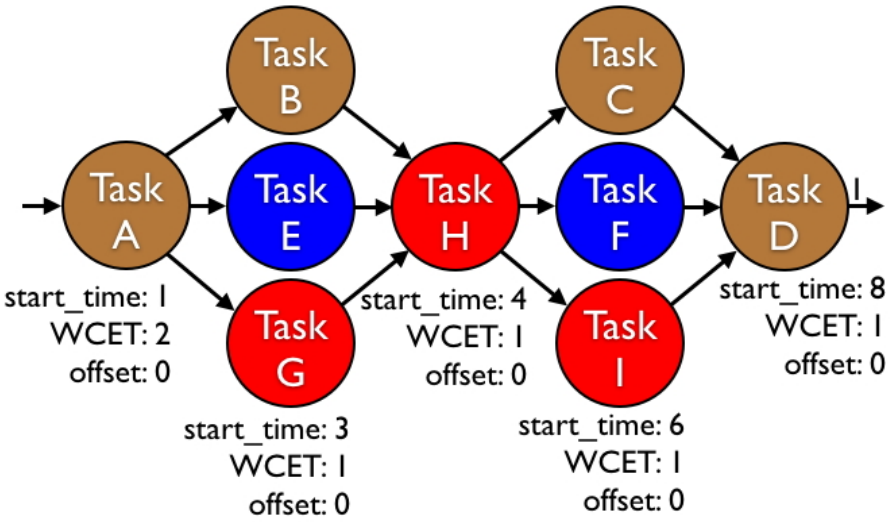

Figure 13: Example Message Connectivity Graph

As expected, Tasks B, E and G, and Tasks C, F and I are in parallel arrangement and have the same start and WCET times since they are members of parallel timing sets. While this example is very simplistic, it is quite possible to have extremely complex task-message connectivity. Each node in the graph has three values: *start_time*, *offset*, and WCET. The start_time and WCET values are taken directly from the TT schedule. Every edge has just one value, mark, initially set at false.

Offset is a calculated value and corresponds to the number of hyperperiods of delay this task will have during execution. Start_time and offset are used in the **ExecTime()** function to calculate the absolute start time of a task's execution. This function is defined as follows:

Algorithm 1 Absolute Execution Start Function

```
ExecTime(node)
    return node.start_time + node.offset * hyperperiod_duration
```

The second step in calculating the response latency is to move through the graph determining the path that causes the longest delay from the initial task to the final task. While calculating the

latency value, it is important to remember that it is possible to span multiple hyperperiods while executing a graph end-to-end, this is where the offset is involved.

The following pseudo-code determines a system's response latency. A global queue of nodes to be processed is maintained and is initially empty. A "source" node is any node that has no input edges, while a "sink" node is one that has no output edges. Functions **SourceNode()** and **SinkNode()** check these conditions with boolean returns. Finally, function **NodeReady()** checks if all input edges to the node have a *mark* value of true.

Algorithm 2 Response Latency Pseudo-Code

```
1:  nodeQueue ← { }
2:  for all( nodes ∈ ϒ )
3:      if( SourceNode(node) ) nodeQueue.push(node)
4:
5:  while( codeQueue.empty = false)
6:      node ← nodeQueue.pop()
7:      max_predecessor = max( ExecTime(all node.inbound_edge.source_node) )
8:      if( node.start_time ≤ max_predecessor.start_time)
9:          node.offset = max_predecessor.offset + 1
10:     else node.offset = max_predecessor.offset
11:     for all( node.outbound_edges )
12:         edge.mark = true
13:         if( NodeReady(edge.destination_node) ) nodeQueue.push(edge.destination_node)
14:
15: last_node = max_forall_nodes( exec_time + WCET )
16: first_node = min_forall_nodes( exec_time for all nodes )
17: Response Latency = ExecTime(last_node) + last_node.WCET - first_node.start_time
```

In our example, the end-to-end response latency is 8ms. Task A is started at 1ms into the hyperperiod, Tasks B, E, G are executed 2ms later, then Task H, Tasks C, F, I, and finally Task D executes at 8ms into the hyperperiod and will run until at most 9ms.

A more complex example may help to illustrate the algorithm. Figure 14 shows a more complex task graph with a hyperperiod of 20ms. Execution of Algorithm 2 results in the shown offsets and an overall response latency of 68ms.

The response latency of a system is perhaps the primary candidate for schedule optimization efforts. Task execution ordering can have a large impact on response latency, especially if the ordering

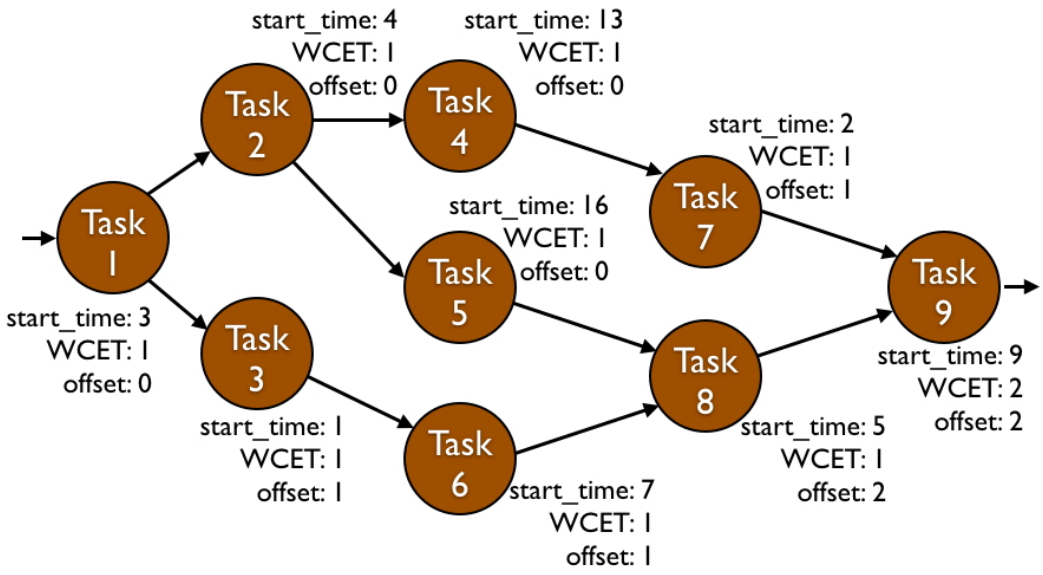

Figure 14: Complex Message Connectivity Graph

causes tasks to execute across multiple hyperperiods. Clearly the system shown in Figure 14 would benefit from schedule optimization because a reordering of tasks through a revised schedule could reduce response latency to as low as 37ms. Additional insight into methods for schedule optimization and response latency reduction can be found in [104].

Clock Synchronization Bounds

Observation & Synchronization Lower Bound

The time-triggered model of computation formalizes system design using time-triggered entities. Fundamental to the time-triggered model of computation is the concept of *temporal-accuracy* of real-time data [105]. First, a few definitions are necessary. A **real-time entity** is something that contains a time-varying internal state that is within the sphere of control (SoC) of the system being designed [106]. An example might be the current spatial position and orientation of an unmanned aerial vehicle (UAV). Each real-time (RT) entity has static properties (e.g. name and type), and dynamic properties, the time-varying state. An **observation** is the capture of a real-time entity's state. It is a tuple of the following form:

$$\text{Observation} = < \text{Name}, \psi_{obs}, \text{value} > \tag{4}$$

An observation consists of the name of the RT entity, the time, according to the local clock, at which the observation was taken, ψ_{obs}, and the value of the RT entity's state when the observation was taken. A **real-time image** is the current understanding of a RT entity. Each observation of an RT entity provides a valid update to the image of that entity for one point in time. The *recent history* at time $\psi_i(RH_i)$ of an image is a set of consecutive observations,$\{\psi_i, \psi_{i-1}, ..., \psi_{i-n}\}$, where the difference in time between ψ_i and ψ_{i-n} is called the *temporal accuracy interval*. Since the entity's state continues to vary over time, any single observation may no longer be valid if the entity's state changes. Therefore, **temporal accuracy** is the relationship between an RT entity and its image. An RT image at time ψ_i is temporally accurate if:

$$\exists \psi_j \in RH_i : \text{Value (RT image at } \psi_i) = \text{Value (RT entity at } \psi_j) \tag{5}$$

The most recent observation in RT image must correspond to the state of an RT entity from within its temporal accuracy interval. This implies that the current state of an RT image may be delayed compared to the current state of its entity, as shown in Figure 15, adapted from [4].

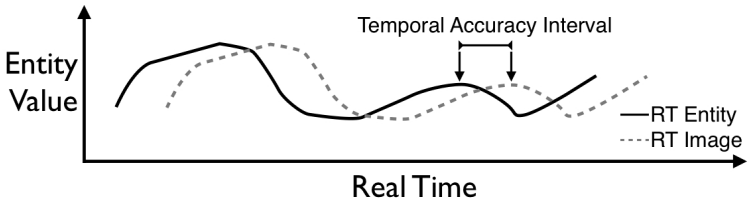

Figure 15: Temporal Accuracy Interval

An important consideration is the amount of delay between when a particular observation is made, ψ_{obs}, and when it is used to affect some change, ψ_{use}. The amount of time it takes to make an observation, communicate that observation, and use it may be greater than the temporal accuracy interval for the RT entity. If this is the case, the state of the RT image when the observation is used must be estimated using a process called *state estimation*.

For example, a simple, though not always accurate, approach to state estimation is to use the first derivative of the state of the entity to estimate its future state. If the value, v, of an observation

is changing over time, $dv/d\psi$, a state estimate can be defined as follows:

$$v(\psi_{use}) \approx v(\psi_{obs}) + (\psi_{use} - \psi_{obs})dv/d\psi \tag{6}$$

The interval defined from when an observation is made to when it is used, $[\psi_{obs}, \psi_{use}]$ is the most important aspect of this equation. Jitter in this interval has a significant impact on the accuracy of the state estimation. In a distributed system it is not uncommon for an observation to be made on one node and used on another. The amount of jitter introduced by the communications network directly impacts the accuracy and performance of the system's output.

Jitter introduced by the network also impacts the ability of a system to synchronize the local clocks across all nodes. According to the impossibility result of [26], given a latency jitter of ρ, it is not possible to synchronize the clocks on N nodes to a greater accuracy than:

$$\Pi \geq \rho(1 - \frac{1}{N}) \tag{7}$$

Where the *synchronization precision*, Π, is in the same units as ρ. The time-triggered architecture demands that all nodes participating in network communication achieve time synchronization within some tolerance. Our definition of local and global clocks, ψ_i and ψ' in a time-triggered system capture this requirement. The impossibility result equation provides a lower bound on the synchronization tolerance given a network's latency jitter, and therefore a lower bound on ϵ.

Determinism & Synchronization Upper Bound

Deterministic behavior is necessary for high-confidence embedded systems if failures are to be handled at the logical level [107] using approaches such as triple-modular-redundancy (TMR). We define *determinism* from [108]:

> A model behaves deterministically if and only if, given a full set of initial conditions (the initial state) at time t_0, and a sequence of future timed inputs, then the outputs and the system state at selected future instants are entailed.

From this definition, variations in behavior can be introduced in two places using our formal model of time-triggered systems:

- Variations in the duration a task takes to execute, its $ExecTime(f_i)$. The upper bound of task execution time is captured in the tasks worst-case execution time, $WCET(t_i)$.
- Variations in the value of local clocks, ψ_i from the global clock, ψ'.

Variations in task duration do not effect determinism under nominal operation. Our task behavioral semantics dictates that all input messages are present at start of a task's execution time. Similarly, all output messages must be present at the end of the task's execution window, $ExecTime(f_t) \leq WCET(t_i)$. As long as the task execution time remains bounded by the WCET, all output messages will be present when they are needed which is at $ST(t_i) + WCET(t_i)$. Variations of the execution time only serve to produce output messages sooner than the end of this time window, which in no way alters future behavior.

Variations on clock synchronization can have a direct impact on the determinism of a system. An upper bound of ϵ, where $\forall\ \psi_i \in \Psi$, $|\psi_i\text{-}\psi'| \leq \epsilon$, was given in the system definition. The actual value of ϵ is closely tied to the determinism of the system. ϵ must be small enough to ensure that the behavior of the system remains deterministic. Per [109], non-deterministic behavior due to clock skew arises from *observation uncertainty*, where two nodes observe the same event but regard it as happening at different times, thus possibly altering event ordering.

In is useful to introduce the notions of a *dense timebase* and a *sparse timebase* [110]. In a system some set of events, $\{E\}$ are of importance, such as the execution of tasks or the sending of messages. If these events may occur at any time during the execution of the system, the system is said to have a dense timebase. If the times at which these events may occur are limited to specific points in time, the system is said to have a sparse timebase. Clearly, a time-triggered system has a sparse timebase as tasks are only allowed to execute at the scheduled start times.

Given a distributed system with a sparse timebase, the occurrence of events must be recognized by all nodes as happening at the "same time". The *global time* of a system corresponds to the virtual clock against which individual timing set clocks are measured. In a sparse timebase the global time

can be approximated via the synchronized ticks of the local clocks. The global ticks must only occur at the scheduled start times for tasks. Each node's local clock must be closely enough synchronized to the virtual global time to allow it to recognize the occurrence of a start-time.

Figure 16 illustrates the global time base for our example and shows the relationship of the global clock tick to the precision of the local clock synchronization, Π. Each period during which events may occur is separated by an interval of silence of some duration, Δ.

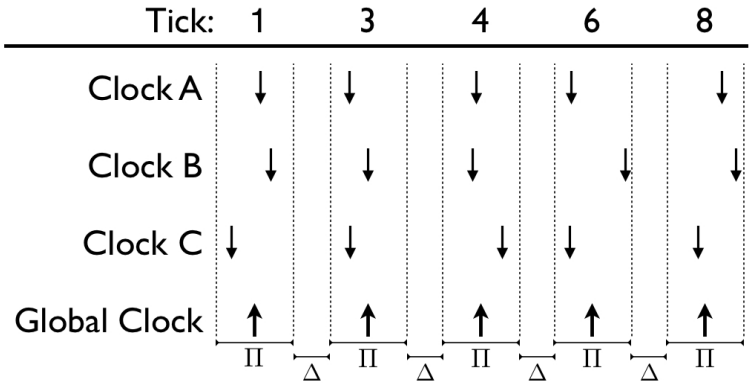

Figure 16: Global Timebase

The precision of the clock synchronization defines the maximum interval over which a local clock may perceive an event occurrence and still understand it to be at a corresponding global tick. The granularity of the global clock, g, must be larger than the synchronization precision: $g > \Pi$. In other words, the synchronization precision must be less than the smallest interval between consecutive task execution starts.

$$\Pi < min(ST(t_{i+1}) - ST(t_i)) \text{ for all } t_i \in T \tag{8}$$

Otherwise, the interval defined by Π at two consecutive global ticks may allow two nodes to recognize one event as belonging in different ticks, and thus introduce non-determinism to the system. This definition of the upper bound on Π gives us ϵ. We now see that Π is bound below by the inability to synchronize distributed clock better that communication jitter allows, and is bound above by the system's execution schedule to prevent non-deterministic behavior.

One final point should be made in this section. Information must flow between the time-triggered system and the environment in which it operates. While the TT system operates on a sparse

timebase, it must be assumed that the external environment has events that may occur at any time and thus has a dense timebase. Some interface between the TT system and its environment must exist in order to mediate the transition from dense time to sparse time and vice versa. A *temporal firewall* [111] acts as a buffer to separate events occurring in a dense timebase from those in a sparse timebase. Given our definition of a TT system, those tasks acting as sources or sinks within a task-message connectivity graph must operate as the temporal firewalls for the system if they interact with the external environment.

Connected System Property

A significant class of real-world time-triggered systems use normal message passing as a means to synchronize local clocks[25] instead of using an explicit clock-synchronization scheme such as PTP [112] or synchronization messages[31, 40]. This scheme is efficient and can be robust to failures, but does require that every node either send or receive messages. This non-timing property can easily be derived through analysis of our formal model.

Definition (Connected System Property): Using our definition any system, Υ, can be transformed to a graph where:

- Graph nodes are the task instances, $t_{i,j} \in T$
- Graph edges correspond to task instance timing set membership. Let all task instances within a timing set, $t_{i,j} \in \lambda_k$, be fully connected.

Υ is said to be a connected system if and only if the resulting graph is connected. Being a connected system is necessary if message passing is the sole mechanism used for local clock synchronization. The graph representation of our example system is presented in Figure 17.

It is clear that our example system is connected and therefore messaging-based time synchronization is possible. As most TT busses are assumed to be broadcast communication medium, a disconnected graph would also indicate a lack of physical connection between portions of the system, thus further indicating a design failure.

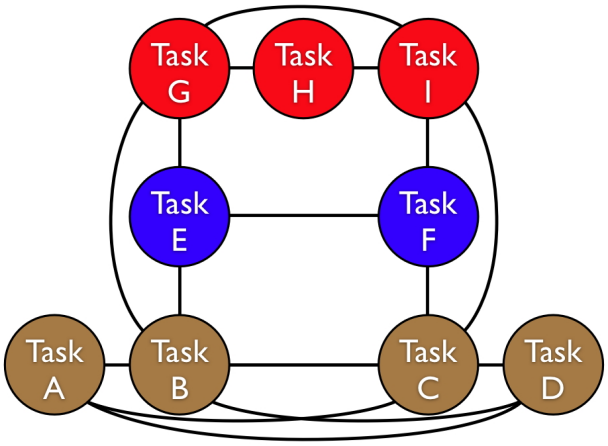

Figure 17: TT System Graph Example

Schedulability & Deadlock

Methods for calculating a valid schedule for a given time-triggered system have been researched in detail [113, 104]. Nearly all of these approaches rely upon the ability to create a weighted acyclic graph of the task execution structure augmented with timing constraints such as was derived above. Current effort is being dedicated to incremental approaches to schedulability analysis and response latency optimization[114] to both improve system responsiveness and better scale schedulability analysis. A detailed discussion of schedulability analysis techniques is beyond the scope of this thesis.

Freedom from deadlock is a natural consequence of the time-triggered approach. As task execution is strictly clock-driven, no tasks may block awaiting input. This places an implicit burden on the TT execution platform to provide native message support as task input messages must always be present. Borrowing from the ARINC-653 specification [50], during execution messages are held in *ports*. Ports may be either *sampling ports*, where the a value is held until it is updated regardless of how many times it is read, or in *queuing ports*, where the most recently updated value may be read only once. For freedom from deadlock, temporal firewalls are best realized using sampling ports. As long as the execution platform is able to guarantee that an executing task will never block trying to read input message or write output messages the system will remain deadlock free.

System Composition

Creation of new systems is often done through joining two existing systems together. Composition of time-triggered systems using our model is straightforward, though not guaranteed to yield a schedulable resulting system. Given two systems, Υ_1 and Υ_2, composition of these systems, $\Upsilon' = \Upsilon_1 \circ \Upsilon_2$, is defined via the individual tuple members:

- $\Psi' = \Psi_1 + \Psi_2$. The local clocks from Υ_1 and Υ_2 are simply combined.
- $\psi' = \psi'_1 = \psi'_2$. The global clock is identical across all systems.
- $M' = M_1 \oplus M_2$. The set of messages and message instances is all messages from both individual systems, but message types with duplicate structure and semantics are consolidated. Message instances are aligned with this, possibly reduced, set of message types.
- $T' = T_1 + T_2$. The set of tasks and task instances is simple addition of the task sets from the individual systems. Interdependence between these sets of tasks depends on if any message types were consolidated. Consolidation implies interdependence between task instances in T_1 and T_2. Start time values, $ST(t_{i,j})$ are not brought through and must be recalculated. Additional discussion of *Sched* is below.
- $\Lambda' = \Lambda_1 + \Lambda_2$. Timing sets are simply additive.
- $Hyperperiod' = LCM(\pi(t_i)) \; \forall \; t_i \in T'$. The new hyperperiod must be recalculated given the composed set of tasks.
- *Sched*′. The worst-case scenario requires a complete reanalysis of the schedulability of Υ' to determine if there is a valid schedule. Related research [114] explores incremental methods to reuse *Sched* information from both Υ_1 and Υ_2 for schedulability of Υ'.

Composition of systems defined using our approach is possible if and only if a valid schedule for the composed system can be found. Other system properties, such as clock synchronization bounds and response latency are directly tied to the execution schedule of the composed system.

Fault Detection & Mitigation

Conformance to a time-triggered schedule imbues a system with robust fault detection characteristics across a broad range of fault hypotheses. Because all message transmissions should occur according to the schedule, any deviation, failure to transmit, transmission at the incorrect time or transmission by the incorrect sender, are easily detected. This stands in contrast to event-triggered networks where the arrival of a message is the only indication that one has been sent and a failure to send can not easily be detected beyond the sending node. In a time-triggered system, failure of a node is detected by other nodes immediately upon a failure to receive a scheduled message. Additional fault-tolerance mechanisms, such as CRC for message corruption, are compatible with time-triggered messaging and can be integrated without compromising fundamentals of the approach.

Numerous fault conditions lie beyond the scope of our time-triggered model but should still be accounted for in an execution platform. For example, TTP/C implements sophisticated group membership management to protect against faulty nodes [25]. Analysis of the 13 assumptions listed in the TTP fault hypothesis [115] reveals that only two are directly tied to the theoretical foundations of time-triggered execution. The remainder are implementation dependent, and relate to factors such as failure rates of particular hardware devices.

Most common fault mitigation approaches, such as TMR, are completely transparent to our time-triggered model. Only approaches that alter system timing, such as message retransmission, are unavailable to execution platform developers.

Conclusion

I have developed a formal model of computation for time-triggered systems. Through analysis of models in this formulation it is possible to derive system properties and characteristics such as response latency, clock synchronization bounds, system connectivity, schedulability and freedom from deadlock.

My time-triggered model is not without compromise, as is true for all time-triggered systems. The global schedule must be created statically based upon the configuration of the system. This works

well as long as the system configuration is static throughout execution. Dynamically evolving systems are becoming more common and adaptation of time-triggered principles is not straightforward. For example, the flight-control system of an unmanned aerial vehicle (UAV) makes an ideal candidate for a time-triggered approach. Its components are well defined, its dynamics are understood, and its configuration is static throughout its nominal flight envelope. However, a "flock" of network controlled UAVs, flying in coordinated flight begins to stretch the applicability of the time-triggered model of computation. UAVs can join or leave the flock at will. Network communications between UAVs will be wireless and subject to unbounded delays and losses. Time-synchronization would be poor. Compositionality in such a system would be an ideal property as UAVs could easily be added or removed while maintaining flock safety, but strictly time-triggered architectures are not yet capable of satisfying such demands. Research on loosening the TT approach's constraints [21, 22] may yield benefits in this area.

An additional compromise made by all time-triggered designs is a trade-off between bandwidth utilization and sporadic events. Take the case of a catastrophic component failure. This failure is highly unlikely to occur but when it does, the entire system should be notified immediately. In a strictly time-triggered architecture, the transmission of this message must occur according to the global schedule. Therefore the global schedule must allocate time for its transmission every hyperperiod even though the probability of using that transmission slot is essentially zero. What should be done if numerous systems have the possibility of sending such high-priority, but low-probability messages? Some combination of time-triggered and event-triggered execution would seem to be needed.

In the following chapters, I will show that from a system defined using my formal model, it is possible to create a practical language for capturing design information, analyzing system properties, simulating system execution, and synthesizing executable code.

CHAPTER IV

ESMOL & THE FRODO V2 VIRTUAL MACHINE

The ESMoL language and its attendant tools attempt to capture the design intent of a high-confidence embedded system into a model. Through analysis and simulation of virtual prototypes ESMoL aims to provide designers with some reassurance about a system's behaviors. Ideally, a toolchain should integrate aspects of the entire lifecycle of designing a system, from initial component specification, through hardware platform definition, to component deployment, model analysis, system simulation, and finally synthesis of executable embedded code. The ESMoL toolchain itself does not try to internalize all of these functions, instead it attempts to provide automated integration with external tools to guide designers throughout the development process.

Several documents have been published by the author and others that cover many, but not all, aspects of the ESMoL language [116, 16, 18, 17] and some of its associated tools [52, 117]. Within the context of this thesis, this chapter details the development and evolution of the ESMoL toolchain, focusing on two particular aspects: the language design rationale and a system model's run-time realization. The core ESMoL language has reached a functional level of maturity and the supporting tools continue to become more complete and better integrated. Public releases of the toolchain with user documentation and example models are available [118].

Example High-Confidence Embedded System

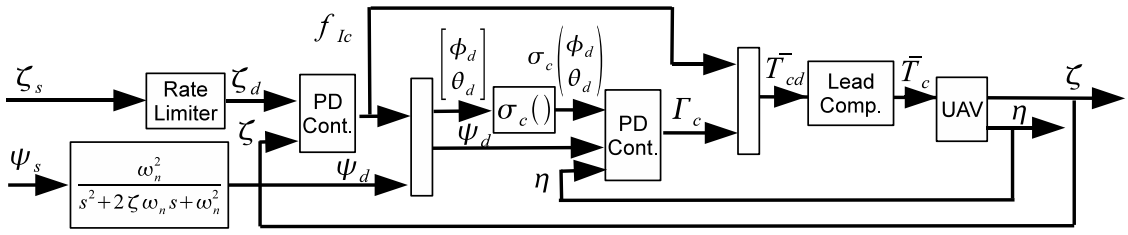

Figure 18: Quad-rotor Controller Architecture [1]

We introduce a new example system, as compared to the simple system introduced in the previous chapter, to illustrate more concrete and complex concepts throughout the remainder of this thesis. This example system is an actuator limited quad-rotor model with a corresponding control architecture [1] as shown in Figure 18. This system's realization in Simulink is shown in Figure 19.

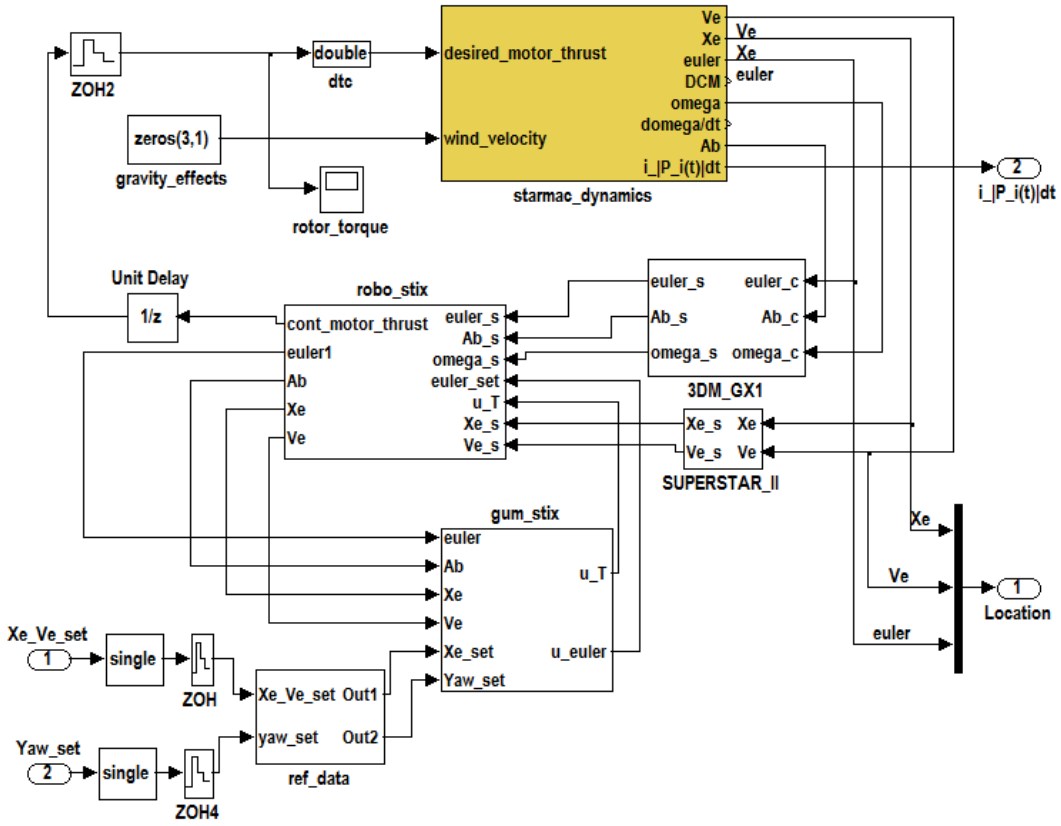

Figure 19: High-level Quad-rotor Controller Model

The flight controller is divided into three primary subsystems: DataHander, InnerLoop and OuterLoop. These subsystems are deployed onto two hardware nodes: robo_stix and gum_stix. The InnerLoop subsystem implements the attitude controller for the quad-rotor aircraft. This subsystem is deployed onto the robo_stix hardware node. The OuterLoop subsystem provides reference point navigation control and is deployed onto the gum_stix hardware node. These two subsystems are separated to reduce the computational load on any one node. The DataHandler block receives the GPS and IMU sensor data and performs some simple unit conversions. It too resides on the robo_stix node. Together these subsystems provide a robust passivity-based control architecture for

the quad-rotor aircraft. This example system will be used frequently throughout the remainder of this thesis.

ESMoL Modeling Language & Tools

The ESMoL language and tools are all based on the Generic Modeling Environment (GME) application [7]. Custom written model interpreters extend the base functionality of the GME environment allowing the ESMoL toolchain to be built up incrementally. These interpreters may import external models, analyze aspects of an ESMoL model, synthesize code or even generate models in other languages. The remainder of this section discusses the steps involved in using the ESMoL toolchain and the primary portions of the ESMoL language. Throughout, we will highlight several of the associated interpreters that are included in the toolchain.

The ESMoL language itself is expressly designed to model time-triggered embedded systems. The formal model for time-triggered systems presented in Chapter III is the foundation upon which ESMoL is created. The primitives available in the ESMoL language are intended to allow designers to capture a system's design intent while maintaining a correlation to the formal model to enable analysis of properties. After a brief overview of the steps involved in creating an ESMoL model, the language and its relationship to the formal time-triggered model of computation will be discussed.

Modeling Steps

The ESMoL modeling tools support the entire process of creating time-triggered embedded systems software, as described in [17]. Before going into greater detail about the ESMoL language it will be helpful to gain an understanding of the steps involved in designing a system using the ESMoL toolchain. The complete process for modeling and generation includes the following steps: (1) Import controller design from Simulink; (2) Define components and message types; (3) Generate component functional code; (4) Create deployment hardware configuration; (5) Deploy components and messages; (6) Calculate time-triggered schedule; (7) Synthesize run-time system code.

The first step is to use an interpreter to import the source controller model from Simulink into the ESMoL modeling environment. The dataflow semantics of the original Simulink model are

preserved and are fully represented within the ESMoL model. Software components in ESMoL are defined by creating references to Simulink subsystem blocks and to input and output message types. Instances of these software components correspond to individual run-time tasks. Each task has logical execution time (LET) semantics [33], in that all input messages are available before a task consuming them is released, and output messages of the task are sent at precisely defined points in time, after the task has finished. Message types and their data elements must also be defined.

Once components and message types are defined, a model interpreter synthesizes platform-independent functional code. The internal dataflow representation of each component is converted into synchronous C-code blocks which will be executed on top of the FRODO virtual machine that implements the TT execution semantics, as will be discussed in the next section. This functional C-code is complied together with a layer of generated "glue code", acting as the TT virtual machine, and together they comprise the embedded software.

The next step is to define the deployment platform hardware configuration. An example deployment platform could consist of a Gumstix Verdex embedded processing module with a Robostix I/O expansion board. These two nodes are connected via an I^2C bus over which time-triggered communications are routed.

Next, instances of software components are mapped onto the platform. This step defines the deployment of software onto hardware. Multiple instances of a particular component may be deployed. Messages are mapped to I/O ports on the hardware to define channels via which they will be communicated. The deployment model is used to determine the configuration of the runtime code.

The second to last step is to execute an off-line schedulability analysis. Automated schedulability analysis begins with the transformation of the ESMoL model into a scheduler configuration file. The configuration file contains an abstract version of the design, limited to information about platform connectivity, assignment of tasks to processors, routing of messages through buses, and timing specifications. The ESMoL static scheduler[52] uses this abstract specification to build a finite-domain integer constraint problem, which is solved using the Gecode constraint logic programming library[119]. Details about the scheduler are documented in [52], though the approach is a refinement

of the constraint formulation first described by Schild and Würtz[120].

The constraint problem models dependencies between tasks and messages, exclusive use of processors and buses, and timing constraints (i.e., maximum acceptable latency between tasks). If the problem is feasible, then a solution will satisfy the specified timing requirements and will contain start times for each of the tasks and messages. Another automated interpreter writes the schedule start times to fields in the original ESMoL model, so that start time information can be used during platform-specific code generation steps. Infeasible problems are reported as such.

The final step in the ESMoL modeling process is to synthesize the "glue-code" necessary for binding the functional tasks to the FRODO run-time platform. Schedule information and task-message connectivity is used to generate all of the information necessary for FRODO to properly execute the system as it has been designed. Glue-code synthesis and the FRODO run-time platform are discussed in greater detail in the following section.

Once a source controller has been imported, components and message types defined, the component functional code generated, a deployment hardware configuration created, components and messages deployed, a time-triggered schedule created and glue-code synthesized, then the ESMoL model is prepared to be compiled with the FRODO virtual machine and executed on a real hardware platform.

ESMoL Meta Model

The core of the ESMoL toolchain lies in the ESMoL domain-specific modeling language (DSML). The DSML is captured in a meta model which defines the universe of elements that a model may be composed of and all valid relationships between instances of these elements. The fundamental elements within ESMoL relate directly to modeling both the software and hardware objects within high-confidence embedded systems, and to our formal model of time-triggered systems.

In the time-triggered model of computation no explicit differentiation is make between computational and communications tasks. While no separation is necessary to support formal analysis, it is more convenient from a user perspective to differentiate between the two. As such, the ESMoL

DSML has the concept of a *Component* for computational tasks and obscures explicit communications tasks within the process of deploying components onto hardware nodes. Figure 20 shows the ESMoL meta model elements related to defining computational tasks.

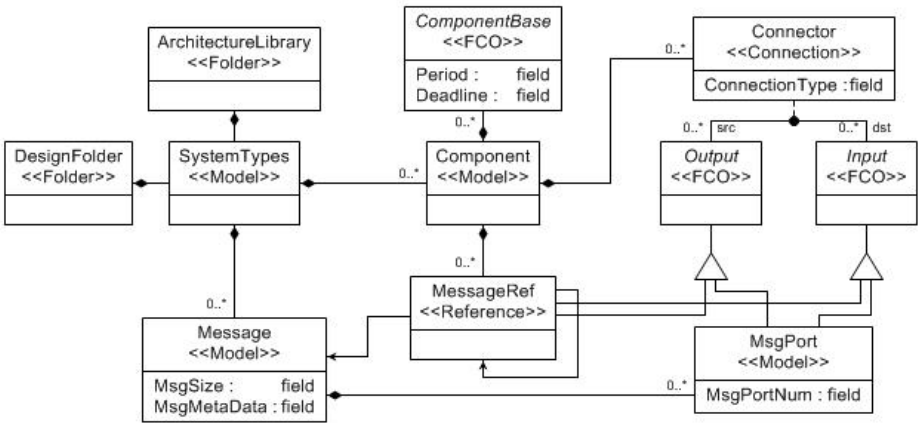

Figure 20: Component & Message Type Definition Meta Model

Message elements in the DSML are a direct embodiment of messages in the formal model, $m \in M$. Message elements contain *MsgPort* elements which are equivalent to members within a structure definition. Each *component* element in the DSML is a computational task, $t_i \in T$, from the formal model. Instances of tasks, $t_{i,j}$, are discussed shortly. Each component (task) has some number of input and output ports which are defined by references to message types. This represents the message instances that are inputs and outputs from tasks in the formal model. In general, it is easy to think of component and message definitions as type definitions in traditional textual programming languages. Numerous instance of these type definitions will be created and used in a system model.

The *connector* element captures the task-message connectivity relationship of tasks producing and consuming message instances. This information is directly used for system connectivity, response latency, and schedulability analysis. With the exception of the *ComponentBase* element, the remainder of the elements in Figure 20 relate to how system models are structured but have no material impact on a system's properties.

The ESMoL language must capture the semantics of Simulink models that are imported into a model. Simulink models are composed of a hierarchy of dataflow blocks combined with stateflow blocks, which are similar in semantics to statecharts [121]. Two sub-languages are embedded within

the ESMoL language for capturing these semantics, one for dataflow and one for stateflow. These two sub-languages are shown in Figures 21 and 22.

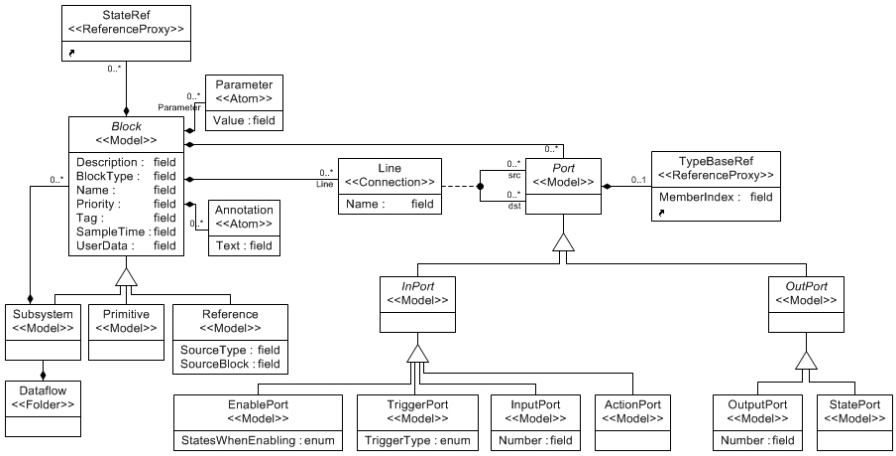

Figure 21: Dataflow Sublanguage

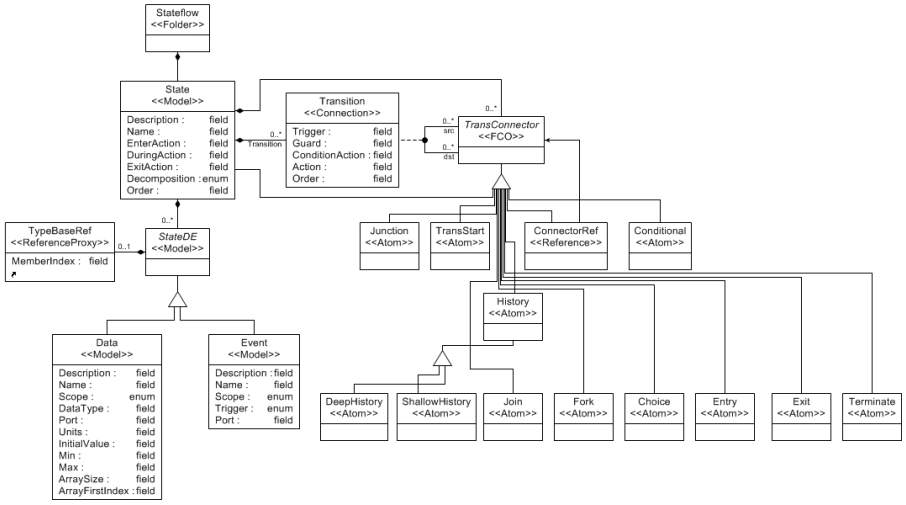

Figure 22: Stateflow Sublanguage

The details of these sublanguages directly mirror the models of computation which they represent. Components, in Figure 20, may contain *ComponentBase* elements which are references to either dataflow *block* or stateflow *state* elements from Figures 21 and 22. Any computational task represented by either dataflow or stateflow elements must adhere to the restrictions placed on all tasks within the TT MoC, namely that they follow LET semantics and are side-effect free. Figure 23 shows the component and message type definitions for our quad-rotor example system. Note how

the internal definition of a component is simply a reference to a Simulink subsystem block.

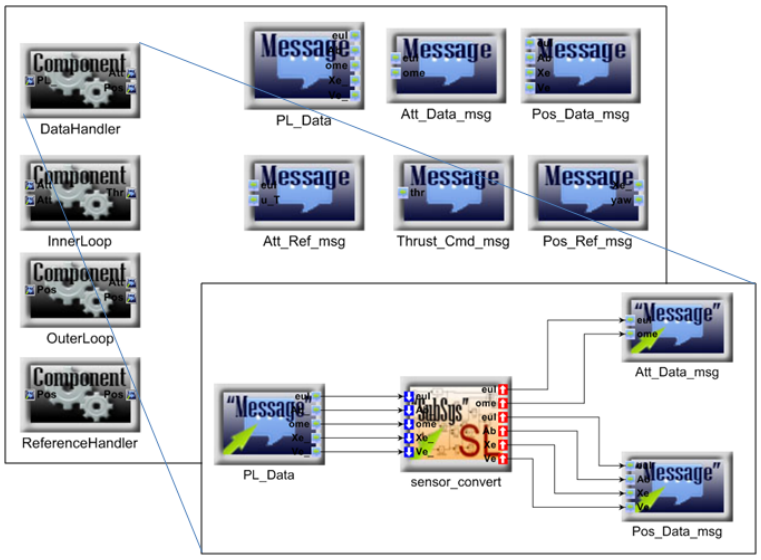

Figure 23: Quad-Rotor Component and Message Type Definitions

Figure 24 shows the elements involved in defining hardware platforms. As would be expected, a *HardwareUnit* is composed of *Node*, *Network*, *IODevice*, and *Bus* elements. Various input, output and bus channel (*IChan*, *OChan*, and *BChan*) communication ports can be added onto a node. It is through these ports that messages will be routed.

As the concept of timing sets from the formal model is quite abstract, we have replaced it in the DSML with several modeling elements that more closely align with how designers perceive systems. Similar to computational tasks versus communications tasks, the hardware platform breaks timing sets into *Node* and *Bus* modeling elements. Both represent an individual unit of hardware, but also directly correspond to a sequential timing set. A hardware platform is composed of a set of one or more Nodes connected via some number of busses.

Figure 25 shows the quad-rotor example's hardware platform with two computational nodes connected via one TT bus. Included in this portion of the model is the *Plant* block which encapsulates the physical dynamics of the helicopter itself. Connecting the plant to the computational nodes are ethernet and UART data channels. These are un-timed (asynchronous) but are still defined using message types.

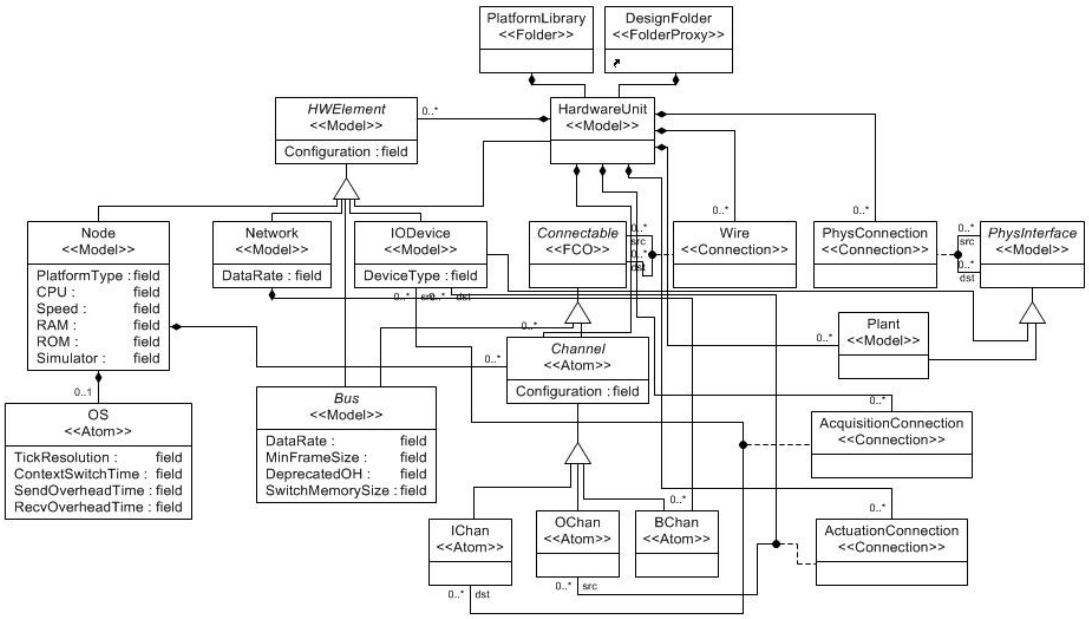

Figure 24: Hardware Platform Meta Model

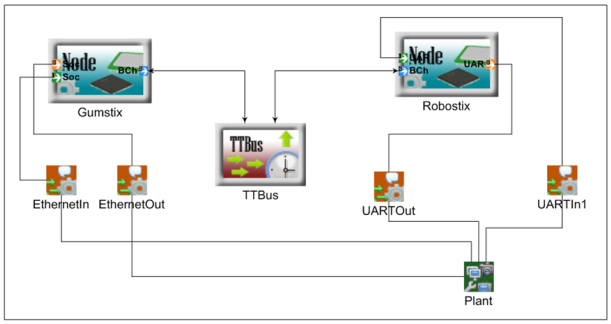

Figure 25: Quad-Rotor Hardware Platform

While the model of computation's abstractions allow it to cover a very diverse set of communications media, the ESMoL language supports only a limited subset due to the necessity of having to provide run-time support for specific network types. The Bus element encapsulates and abstracts all of the detailed information necessary to model the actual bus hardware. For example, in the quad-rotor example the two nodes are connected via and I^2C bus. The system designer must only select I^2C from an enumerated list of supported network types and provide the maximum data rate to fully specify the bus. The fact that I^2C busses act as *parallel timing sets* between senders and receivers is abstracted from the designer and is managed by the ESMoL tool suite itself. Similarly, if an full-duplex ethernet network was specified it is represented by a *sequential timing set* between

senders and receivers.

Hardware platforms can be modeled completely independent of software components and message types. As can be seen in Figure 24 no relationships can be established between hardware elements and software components at this stage. Instead, interactions between instances of components define a separate concept, the *logical architecture* of a system. Figure 26 depicts the portion of the ESMoL meta model related to capturing the logical architecture of a system. A *ComponentRef* is a reference to a component, or in other words, an instance of a task, $t_{i,j} \in T$ from the formal model. Similar is true for *MessageRef* objects and messages, $m_i \in M$. A *System* is composed of component instances passing message instances to other component instances. Once a logical architecture is defined, deployment of components onto hardware and timing set membership for tasks are captured using the *ComponentAssignment* relationship between components and nodes and the *CommMapping* relationship between messages and communication channels.

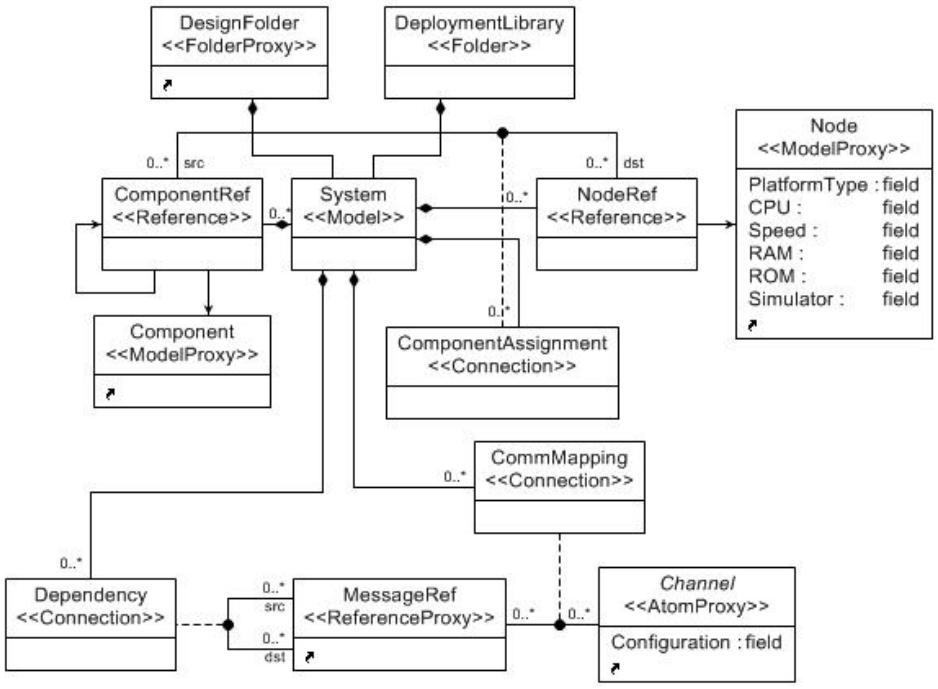

Figure 26: Logical Architecture & Deployment Definition Meta Model

Using the logical architecture, composed of task instances producing and consuming message instances, it is possible to build the task-message connectivity graph for the system. From the

graph, analysis, such as response latency, can be conducted. Figures 27 and 28 show the example quad-rotor component logical architecture and their hardware deployment mapping respectively.

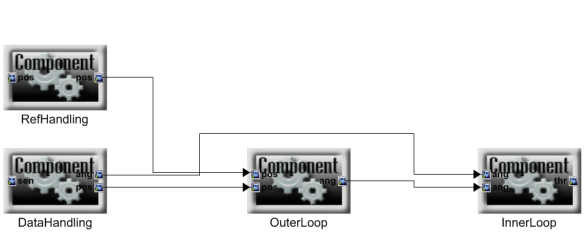

Figure 27: Quad-Rotor Logical Architecture

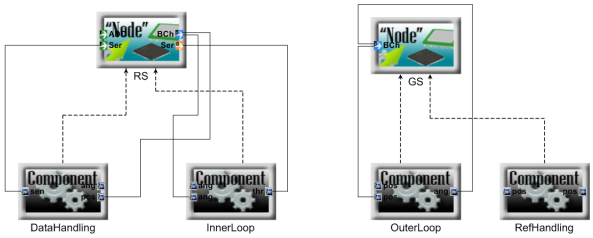

Figure 28: Quad-Rotor Component Deployment

Using Figure 27 it is easy to see that an instances of an OuterLoop components receives messages from the instances RefHandling and DataHanding components while the InnerLoop instance then receives messages from both OuterLoop and DataHandling. In the logical architecture no notion of hardware is needed, just simple task-message connectivity and ordering. Figure 28 shows how the component instances are deployed onto the hardware platform. Message inputs and outputs for each component must be mapped to specific message ports on the hardware which in turn correspond to connectivity via some communications system.

Because the ESMoL modeling language has been derived from the formal model of time-triggered systems, a unique formulation of the system as an abstract model is always possible. With the formal model in hand, other forms of analysis are possible. There are other elements within the ESMoL meta model but the portions discussed above constitute the central modeling language.

ESMoL Schedulability & Response Latency Analysis

From a fully defined ESMoL model, automated tools support the schedulability analysis of a system. The result of such analysis is either a valid schedule or an error alerting the designer that no possible schedule exists. The first step in this process is the generation of the schedulability analysis input file. This file defines all of the entities that need to be scheduled and their parameters. Figure 29 shows this file for the quad-rotor example.

The file is broken into four parts: the resolution header, and three sections, one section for

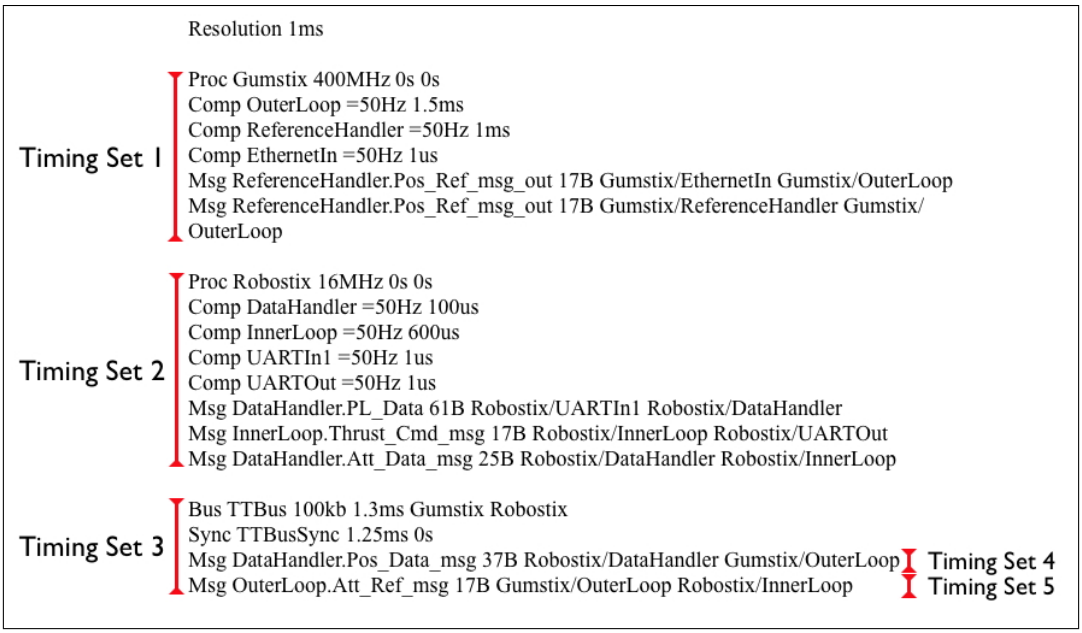

Figure 29: Quad-Rotor Schedulability Analysis Input

each node and one for the bus. The system resolution is the time increment on which start times are aligned. The three groupings mirror the relationship of tasks to their timing set. Each node constitutes a sequential timing set as does the bus, yielding timing sets 1, 2 and 3. Timing sets 4 and 5 are parallel sets for two separate message transmissions. The sender and the receiver are specified as the bus participation is evident.

From this file a valid execution schedule is generated using the ESMoL scheduling tool [52, 114]. The whole system operates with a hyperperiod of 20ms. Table 2 shows all of the devices, their assigned task instances, and the start time for each instance. The WCET for each task was determined experimentally and manually annotated in the ESMoL model.

Table 2: Quad-Rotor Example Schedule (in ms)

Device	Task Name	Start Time	WCET
Robostix	UARTIn	3	$1\mu s$
Gumstix	ReferenceHandler	5	$1ms$
Gumstix	EthernetIn	6	$1\mu s$
Robostix	DataHandler	6	$100\mu s$
Robostix	InnerLoop	8	$600\mu s$
Robostix	UARTOut	10	$1\mu s$
Gumstix	OuterLoop	10	$1.5ms$
TTBus	OuterLoop.Att_Ref_msg	12	$1ms$
TTBus	DataHandler.Pos_Data_msg	16	$1ms$

The ESMoL scheduler uses task-message connectivity information to create a partial ordering on task execution and can identify opportunities for parallel execution. In Table 2, the EthernetIn task on the Gumstix executes at the same time as the DataHandler task on the Robostix. Using the task-message connectivity information in conjunction with the fully specified execution schedule, end-to-end response latency can then be calculated. Figure 30 shows the quad-rotor's graph and its response latency analysis. There are two hyperperiods worth of delay using the provided schedule, resulting in 47ms of total response latency. One important thing to note is that the parallel sets of tasks corresponding to TT message communications have been condensed in this figure into blue "TTBus" tasks for visual simplicity. The ESMoL model and tools maintain the full data necessary to capture these relationships.

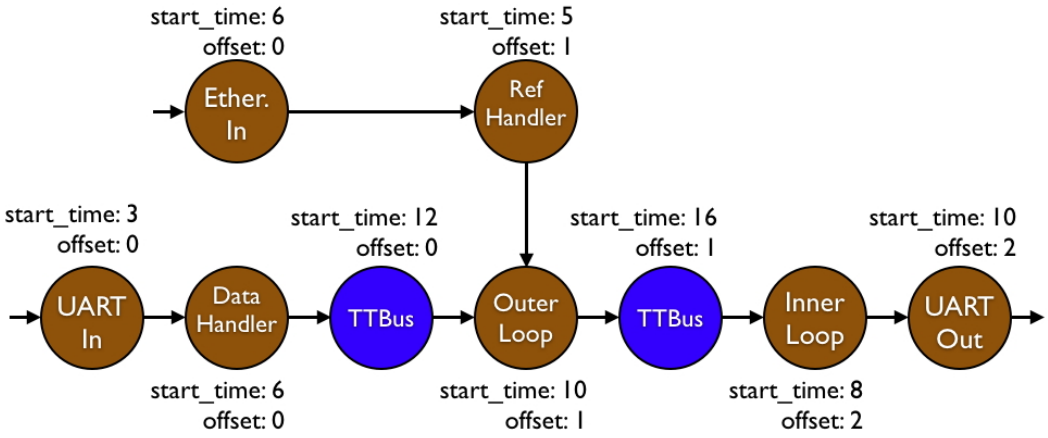

Figure 30: Quad-Rotor Response Latency Analysis

FRODO v2 Virtual Machine Implementation

The process of synthesizing an ESMoL model into deployable and executable code is a significant challenge. The functional code necessary to implement the software components is generated directly from the Simulink blocks referenced in a model, but the time-triggered behavior assumed by ESMoL and its tools must also be realized. Time-triggered execution could be supported directly at the operating system level, but few such systems are available [122, 123]. Instead, an alternate approach is to implement a time-triggered execution layer on top of standard OS primitives [33, 43],

as shown in Figure 31. As long as all underlying time-triggered assumptions are maintained by the execution layer, this approach provides a portable and light-weight platform for robust execution. The FRODO v2 virtual machine (VM) provides time-triggered execution semantics for ESMoL models and has been tightly integrated into the overall ESMoL toolchain with tools to automatically generate all needed "glue code" to create a functional executable.

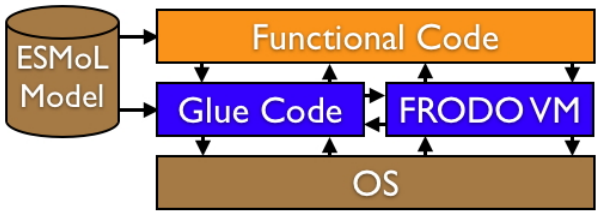

Figure 31: FRODO v2 Conceptual Architecture

The requirements for a time-triggered virtual machine are focused on two areas. First, the VM must be capable of scheduling and executing computational tasks on a single node such that they adhere to the static global schedule. Through the use of high-resolution clocks and timers present on most modern CPUs this area is the easier of the two to accomplish on operating systems that have configurable thread preemption schemes, such as Linux. The second focus area is message passing and clock synchronization between nodes. Ideally, the VM should support multiple network types and should be robust to faults and stochastic transmission delays. Variable levels of support for different media across OSs, multiple media access API standards, and myriad avenues in which to introduce faults make this area more difficult. The FRODO v2 virtual machine was designed from the start to tackle both of these areas while maintaining tight ties with the ESMoL language and toolchain.

The ease of experimentation with the FRODO v2 virtual machine has been an additional research goal. The FRODO VM supports execution on multiple OS platforms, several different communication network types, and provides fully-integrated data capture and analysis facilities. Because of FRODO's tight integration with ESMoL toolchain, and the fact that the toolchain fully synthesizes all necessary code, creating and deploying an experiment using FRODO is trivial once an ESMoL model has been fully defined. A designer can easily create a model in Simulink, annotate it in

ESMoL, generate a complete FRODO-based executable, and run experiments entirely on his individual machine. The ease-of-use and flexibility of ESMoL combined with FRODO has allowed us to complete several experiments in a relatively short timeframe compared to traditional development approaches.

Maintainability and portability across a range of host OSs is also an important consideration for FRODO. The VM is implemented as a low-level, portable C-code library with OS-dependent primitives such as threads, semaphores, and signals isolated into an OS-abstraction layer. The VM runs well on top of a number of existing operating systems, including: Apple OS X 10.5 and 10.6, Microsoft Windows 7 and XP SP3, Linux 2.6.22+, and FreeRTOS 5.x [124]. The following subsections discuss the key aspects of FRODO VM, considerations taken during implementation and the associated contributions.

Several notable issues had to be overcome during the development of FRODO. Foremost among them, our desire to have both high timing accuracy and OS/platform portability were often in conflict. Each OS provides unique interfaces and semantics for interaction with its clock and timer primitives. A simple monotonic clock abstraction had to be assumed by the FRODO architecture and the run-time implementation for each platform had to be extensively hand-tuned in order to optimize timing accuracy. The process of porting to a new operating system requires a deep understanding of timing and scheduling issues within that OS and how to best exploit them. Another major challenge was the API design of FRODO itself. Our goal was to keep the interface flexible enough to meet ESMoL's needs yet simple enough to minimize the code base and therefore possible errors. Our effort developing the TrueTime-based time-triggered VM (see Chapter V) provided the necessary experience to guide us through the FRODO API design.

Online Time-Triggered Scheduler

The heart of the FRODO v2 virtual machine lies within its online scheduler. If a valid schedule is found via the static analysis tool, see ESMoL modeling process step 6 above, the ESMoL toolchain uses templates to synthesize all necessary functional and "glue" code for a FRODO-based executable. The glue code consists of FRODO API calls that configure the VM environment (e.g. scheduler,

communication channels, message ports, logging), and creates tasks and messages with associated scheduling information. During code synthesis, all scheduling information is embedded into the generated glue code itself (i.e. no external file dependencies since some platforms do not have accessible filesystems).

Table 3: FRODO API Summary

Function	Description
SchedInitialize	Initialize the online scheduler. Takes hyperperiod duration as a parameter.
SchedShutdown	Shut down the online scheduler and therefore the whole system.
SchedCreatePeriodicTask	Create a time-triggered task. Takes start times as a parameter.
SchedExecute	Begin execution of the time-triggered schedule.
SchedSignalExecution	Function called by each task to await execution.
SchedSignalCompletion	Function called be each task at end of execution.
UDPInitialize	Initialize the UDP communications system.
UDPShutdown	Shut down the all UDP channels and clean up the UDP system.
UDPCreateChannel	Create a new UDP communications channel. Takes transmission times with send/receive ports.
SysEventInitialize	Initialize the event logging system.
SysEventRegisterCategory	Register a category with which log events may be submitted.
SysEventRegister	Register a sub-categorical type with which log events may be submitted.
SysEventSetMask	Enable/disable logging with specific event categories and types using bit-mask.
SysEvent	Log an event. Takes event category and type as parameters.
SysEventsProcess	Process all buffered events. May be called at shutdown or at each hyperperiod end.
SysEventShutdown	Shutdown the event logging system.
CreateSamplingPort	Create an ARIC 653-style sampling port.
WriteSamplingMessage	Write a message into a particular sampling port.
ReadSamplingMessage	Read the message in a particular sampling port.
GetSamplingPortID	Get the ID of a named sampling port.
GetSamplingPortStatus	Get the status (freshness) of a sampling port.

Figure 3 summarizes the FRODO API. Most of the functions are self-explanatory. Overall, the API closely matches the TrueTime v2 Beta 6 API [125]. This was done purposefully to minimize differences between FRODO code and templates and those for our ESMoL TrueTime tool, see Chapter V. Figure 32 shows the primary FRODO glue-code generated for the RoboStix node in the Quad-Rotor example (all times are in ms).

First, on a per-node basis, the scheduler is initialized with a $20ms$ hyperperiod. Next, task instances are created and given an execution schedule. The entry point into the FRODO scheduler

```
// Initialize Scheduler with hyperperiod length and context
FS_Initialize( 20.0, 1, &context );

/*** Create Tasks ***/

// Create RefHandler task with instances
pfloat_t RefHandler_times[1] = { 5.0 };
FS_CreatePeriodicTask( "RefHandler", RefHandler_exec, &context.RefHandler, 0.001, NoRestriction, 1, RefHandler_times );
// Create OuterLoop task with instances
pfloat_t OuterLoop_times[1] = { 10.0 };
FS_CreatePeriodicTask( "OuterLoop", OuterLoop_exec, &context.OuterLoop, 1.5, NoRestriction, 1, OuterLoop_times );

/*** Create Peripheral Channels ***/

pfloat_t Receive_times[1] = { 12.0 };
pfloat_t Send_times[1] = { 16.0 };
FP_SyncExpectation expects[2] = {
    { ExpectReceive, 1, sizeof(UDPReceive_msg_t), 1, Receive_times, 1.0, &context.udpReceive_msg, context.udpReceive_sem, NULL },
    { ExpectSend, 1, sizeof(UDPSend_msg_t), 1, Send_times, 1.0, &context.udpSend_msg, context.udpSend_sem, NULL }
};
context.udpChannel = UDP_CreateChannel( UDP_LOCALADDRESS, UDP_BROADCAST, 21212, 2, expects, 0, NULL, true, true, 2 );

/*** Execute the schedule ***/

// Wait for the platform to wake up ( Platform Dependent )
NanoSleep( SCHEDULER_INITIALIZATION_WAIT, SCHEDULER_RESOLUTION_MS );
// Execute the schedule
FS_Execute( );
```

Figure 32: FRODO Glue Code for the Quad-Rotor Example

for computational tasks is the `CreatePeriodicTask()` function. This function takes as parameters: a task name, a function pointer for the code to be executed each invocation, a task context for any persistent data, the task WCET, and an array of execution start times within a hyperperiod. The *RefHandler* and *OuterLoop* tasks each have one instance. *RefHandler* is scheduled to execute at $5ms$ into each hyperperiod with a WCET of $1\mu s$. Similarly, the *OuterLoop* task begins executing at $10ms$ into the hyperperiod and has WCET of $1.5ms$. Within FRODO an OS thread is created for each task as well as one for each message transmission expectation. Each thread has time-triggered execution information associated with it, and all threads are managed by the FRODO scheduler. After tasks the TT communications are established.

Message passing is similarly defined using the `CreateChannel()` function. A key structure called a `SyncExpectation` captures the expectation of a message being either sent or received on a given communication channel. Each expectation defines which message type will be passed, in which direction it will travel, how long transmission should take, and at what time the transmission should initiate. Each inbound or outbound message is also associated with a ARINC 653-style sampling port. For the quad-rotor, one message is expected to be received at $12ms$ and is placed into the **context.udpReceive_msg** port, while one outbound message is sent at $16ms$ and is taken from the **context.udpSend_msg** port. Any computational task needing access to the inbound message or generating the outbound message must simply read from or write to the correct port. The transmission expectations are associated with a specific communications channel. In this case, a

UDP socket over ethernet that broadcasts on port 21212.

Finally, the scheduler is told to execute the schedule. Both nodes in the quad-rotor system reach this point at typically different points in time. Obviously they must synchronize with each other in order to begin coordinated execution of the schedule. FRODO supports a simple three-way coordinated initialization protocol that is robust to node failures. First, a function is defined, **min**(*node.address*) for each network type. The node with the lowest **min** value is "elected" the initialization master. All other nodes announce their presence and wait for an acknowledgement from the master. Once the master has acknowledged all "slave" nodes, it sends out a "firework" message to signal the start of the first hyperperiod. Every node gains a rough understanding of the round-trip message time between itself and the master through the announce-acknowledge process, and it uses this information to compensate for transmission delay in the firework message. If the node selected as master does not acknowledge an announcement or does not send out a firework message within certain timeout values, the remaining nodes assume it is faulty and elect the next node according to the min function.

```
// Cycle forever, or until _schedExec is set to false
while ( _schedExec ) {
    // Is this the start of a hyperperiod
    if ( hyperperiodStart ) {
        // Clear the start flag
        hyperperiodStart = false;
        // Clear the clock (first time desired == Sync delay, then hyperperiod for the remainder)
        ZeroTime( desiredMS );
        // Set desiredMS to hyperperiodMS (see previous comment)
        desiredMS = _hyperperiodMS;
        // Find the next task
        taskStartTimeMS = _GetNextTask();
    }
    // Otherwise, execute the ready task
    else {
        // If there is a next task ready to run
        if ( _nextInstance != NULL ) {
            // Set the schedulable instance
            _nextInstance->schedulable->execInst = _nextInstance;
            // Signal the task to run, only takes a post to the task semaphore
            _PostSemaphore( _nextInstance->schedulable->semaphore );
            // Check for WCET overrun
            _CheckTaskOverrun( _nextInstance );
        }
        // Find the next task
        taskStartTimeMS = _GetNextTask();
        // Double check for end of hyperperiod
        if ( taskStartTimeMS == _hyperperiodMS ) {
            // Mark the start of a hyperperiod
            hyperperiodStart = true;
        }
    }
    // Sleep until the next known event (task execution)
    NanoSleep( taskStartTimeMS );
}
```

Figure 33: FRODO Online Scheduler Code

The FRODO scheduler always gets its own OS thread which has the highest (i.e. able to preempt

all other threads) priority. Figure 33 shows the core FRODO scheduler code. At the beginning of each hyperperiod the scheduler thread awakens and determines the time at which the first time-triggered task or message transmission will occur. An OS-dependent implementation of `NanoSleep()` is used to sleep the scheduler until this time. The scheduler reawakens and immediately releases the task or message via a semaphore. The scheduler returns to sleep until the WCET of the task or message should have elapsed. When a task completes its execution it notifies the scheduler using a semaphore. Upon awakening, the scheduler checks for this notification. If present, the scheduler moves on to the next task. If not present, an overrun error has occurred and a mitigation effort is undertaken. The scheduler continues to execute until the system is halted or if a unrecoverable fault occurs.

Network Communication Support

Inter-node time-triggered communications is a fundamental aspect of FRODO. Full support for UDP-over-ethernet based communication channels has been built into the virtual machine. The functionality of this channel closely mirrors that of TTP/Ethernet [29]. Messages are passed on a strictly static schedule and any faults are easily detected. Additional channel services, such as execution initiation coordination and time synchronization, are also supported.

Within FRODO there is emerging support for I^2C and serial communications channels. I^2C is different from ethernet in that it does not provide full message buffering on both the sender and the receiver. As a result, the sending and receiving threads must be fully synchronized and will both block until the message is fully transmitted, where as in ethernet the sending thread only blocks while the message is DMAed from the processor to the ethernet controller and the receiver is only notified once the full message is received. This difference in semantics is easily captured in our abstract TT model of computation using either parallel or sequential timing sets to represent the communications bus. I^2C acts as a parallel timing set while ethernet acts at a sequential timing set. Depending on the bus type selected in ESMoL, the appropriate schedule and FRODO configuration are derived. Serial busses have the flexibility to act as either parallel or sequential timing sets and depend upon the designer to specify the correct configuration.

Time synchronization between nodes in a time-triggered systems is critical. Synchrony between

nodes must be maintained as a fundamental precept of a TT approach. Through synchrony, error conditions may be detected and correct execution maintained. As with TTP/C and TTP/Ethernet, synchrony is accomplished as a result of message passing. Each node knows when every message transmission should occur through the creation of an expectation. Upon receipt of a message the scheduler calculates the offset between the expected receipt and the actual. At the conclusion of every hyperperiod, all of these offsets are accumulated and averaged using Lyndelius-Lynch [26]. This average, which is shown to converge across a network of nodes over time, is used to slightly adjust the internal clock of each FRODO node.

Error Detection & Mitigation

Robust fault detection and mitigation are two of the most important theoretical advantages of time-triggered architectures. Because of the strict timing and synchronization imposed on tasks and messages, time-based faults are easily detected. FRODO is able to detect all major classes of fault condition including: task execution overrun, message transmission failures, node failures, and unexpected or incorrect transmissions. For each of these cases a default mitigation strategy is provided, but a customized strategy is easily configured for individual fault conditions.

In Section 4 of [42] the author compares the fault management capabilities of TTP/C and FlexRay across numerous fault scenarios. This is not intended to be a comprehensive list of all possible fault scenarios, but instead was meant to highlight TTP/C's better fault management properties in specific scenarios where FlexRay was less capable. We have extended their analysis to include FRODO and its capabilities across these fault scenarios. A detailed description of each scenario is included in [42].

Table 4: FRODO, TTP/C, FlexRay Fault Scenarios

Scenario	**FRODO**	**TTP/C**	**FlexRay**
Outgoing Link Failure	Yes	Yes	No
Slightly-off-Specification	Yes	Yes	No
Spatial Proximity Failure	No	Yes	No
Masquerading Failure	Yes	Yes	No
Babbling Idiot Failure	No	Yes	No

In Table 4, a **Yes** indicates that the protocol is able to detect and possibly mitigate this fault condition. A **No** indicates that the particular fault scenario is either not properly detected or the protocol provides no possible mitigation path. From Table 4 is can be seen that FRODO falls roughly between TTP/C and FlexRay in terms of identifying and attempting to mitigate these faults.

FRODO Testing & Benchmarking

Automated testing and benchmarking play an important role in validating FRODO execution as it is ported to different OS platforms and helps to ensure consistent performance. A suite of automated unit, functional, and temporal benchmarking tests are executed against the FRODO library on each supported platform.

Timing benchmark tests have been conducted for several host OSs (Apple OS X 10.6, Microsoft Windows XP SP3, and Linux 2.6.34 both on a typical desktop machine and on a tuned embedded system) to ascertain the extent of timing variability introduced into strictly time-triggered execution by the underlying OS's themselves. For each platform, a test of 1,000,000 tasks and 1,000,000 message sends and receives was executed. Given a 20ms hyperperiod and nine time-triggered executions per hyperperiod, this corresponds to roughly 12.3 hours of execution time. The absolute value of the offset between the actual and the expected execution times were logged. The results are shown in Table 5.

Table 5: FRODO Host Platform Timing Characteristics

	OSX 10.6.4	**Win. XP SP 3**	**Linux 2.6.34 (dt)**	**Linux 2.6.34 (em)**
Min. (μs)	<1	<1	<1	47
Max. (μs)	4747	7746	40	54
Avg. (μs)	35	44	32	50

As expected, the two user OSs, OSX and Windows, showed extreme variability in their ability to meet the TT schedule, with greater than $4s$ and $7s$ of difference between min and max respectively. The two Linux tests were conducted using nearly identical OS configurations but on two different hardware setups. The (dt) test was conducted on a typical desktop machine that included power management in the BIOS. The (em) test was conducted on a small Atom powered machine that

provided no processor scaling or power management. The Linux kernel, using the RT_PREEMPT [126] patch, provides fine-grained control over task scheduling and allows everything except for power management interrupts to be preempted.

The results of these tests serve several purposes. First, they reinforce the necessity of deploying a TT-focused executable on top of a embedded real-time oriented host OS. Due to its more configurable scheduler, Linux fared much better in both the maximum and average execution jitter as compared to the non-configurable consumer-oriented OSs. This directly relates to the time synchronization bounds property calculated using the formal model. Clearly a host platform timing variability in the > 4000 μs range is not acceptable for tasks spaced $1ms$ apart. Second, if ongoing development of the FRODO VM were to introduce unexpected or unacceptable timing characteristics, these would be quickly identified through the benchmarking and testing suite and remediation could more easily take place. Finally, as FRODO is ported to new operating systems, the suite of tests provides a measure of the maturity and correctness of the virtual machine on that host platform.

FRODO Experimentation

A number of experiments have been conducted utilizing the FRODO v2 virtual machine in combination with the ESMoL toolchain. Each experiment starts similarly, a source model is imported into ESMoL from a 3^{rd} party modeling tool, namely Simulink. Once imported, the model is annotated with a deployment hardware platform, software components and messages are defined, a deployment is created, schedulability analysis conducted and finally code synthesized. The process to generate, compile and deploy code has been automated to the extent that takes a few minutes at most. This compares well to existing approaches which typically take much longer.

Conclusion

In this chapter, I presented the ESMoL modeling language and its associated tool suite. Models defined in ESMoL capture the complete design of time-triggered embedded systems as defined by the formal TT model of computation. The supporting tools automate and streamline the system development process greatly reducing the workload of designers and helping to ensure a more correct

design. Also presented was the FRODO v2 cross-platform, time-triggered run-time framework. It has robust support for fault detection and recovery, and the ability to execute reliable messaging across a range of communication media.

In order to validate ESMoL and FRODO, we created an automated testing and benchmarking suite that checks the correct execution and performance of the FRODO virtual machine across a range of host operating systems. This suite is tightly correlated to the development process and guides the reliability and performance expectations as FRODO is used on various OSs. Multiple experiments were performed using the FRODO v2 virtual machine to validate the real-world performance, reliability, and effectiveness of our implementation. These experiments demonstrated the efficacy of our approach and showcased the ability of the ESMoL toolchain to quickly allow designers to create robust embedded systems.

CHAPTER V

TRUETIME MODEL SYNTHESIS

Designs for embedded control systems typically start with an "idealized", or time-invariant, controller model. This model usually does not take into account the real-world hardware environment onto which the controller will be deployed. Deployment of the controller onto actual hardware often introduces temporal effects which may degrade performance or alter the expected behavior of the controller. Temporal effects can stem from constraints imposed on the controller by the hardware, such as from limited CPU capacity or inadequate communications bandwidth, or even from the specific task scheduling algorithm used.

Current state-of-the-art model-based controller development environments, such as Simulink/ Stateflow[68], do not directly support the concept of a deployment platform and do not natively simulate the impact of deployment on controller performance. Third-party extensions to Simulink have been developed that allow these impacts to be simulated and analyzed. The TrueTime toolbox[11, 69] is a third-party suite of Simulink blocks designed expressly for this purpose. TrueTime supports modeling, simulation, and analysis of distributed real-time control systems including real-time task scheduling and execution, various types of communications networks, and "analog" inputs and outputs for interaction with the continuous-time plant model. While gaining insight into platform effects is crucial, TrueTime imposes an additional burden on systems engineers. It requires significant effort and a deep understanding of both TrueTime and the desired deployment platform in order to adapt time-invariant models into TrueTime models.

TrueTime's flexibility allows for it to model a wide range of real-time platforms, from simple systems through complex hard real-time architectures. The time-triggered architecture has been shown to provide the necessary services to create robust, fault-tolerant control system communications. In our interpretation of time-triggered control systems some of the key architectural requirements are statically scheduled task execution, tight time synchronization between nodes, strongly controlled time-based bus access, and robust support for identifying and handling fault conditions. The TT

approach provides a fully synchronous distributed environment, at the possible cost of additional time delays between distributed functions. The TrueTime toolbox's primitives have all of the necessary features required to support these concepts, but it does not directly implement a time-triggered platform.

We have developed an extension to the ESMoL tool chain that automatically synthesizes a time-triggered TrueTime model from a design description captured in the ESMoL language for the purpose of simulating and analyzing the impact of platform effects on embedded control systems. Using the TrueTime model possible deployment flaws, schedule timing inconsistencies, and controller stability or performance issues can be identified far in advance of deploying the controller onto real hardware.

TrueTime Overview

The TrueTime toolbox is comprised of a set of Simulink modeling blocks, shown in Figure 34. These blocks integrate with existing Simulink blocks and allow designers to simulate hardware nodes executing tasks using a variety of scheduling policies and communication network types. Out of the TrueTime toolbox, only the *Kernel* and *Network* blocks are of interest to us.

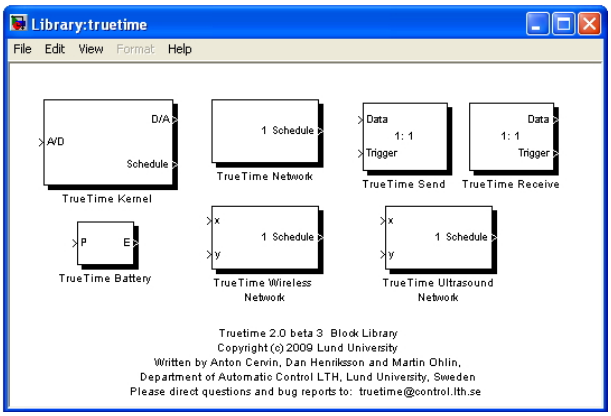

Figure 34: Modeling Blocks Available in TrueTime

A TrueTime kernel block represents a set of tasks executing on one hardware node. They interact with the rest of the Simulink model via "analog" inputs and outputs. Interactions directly between kernels are communicated using the TrueTime network block. Each kernel also generates a standard Simulink-style plot of showing task execution over time.

The TrueTime network block abstracts a digital communications network between kernel blocks.

Each network block represents an individual network and is configured from one of many networks types: CSMA/CD, Round Robin, FDMA, TDMA, Switched Ethernet, FlexRay, and PROFINET. Each of these network types must be configured with bus speed, frame size, loss probability, etc. Currently, the TrueTime network block does not support either I^2C or Serial communications.

A mapping between our abstract time-triggered model of computation and TrueTime needed to be found in order to support time-triggered execution within TrueTime. A kernel block corresponds to a set of sequential computational tasks, and therefore a sequential timing set, from our abstract TT model of computation. The TrueTime network block only supported fully buffered and duplexed network types and as such is also represented as a sequential timing set.

ESMoL Model Specification and Toolchain

Before any TrueTime models can be synthesized, a complete system must be specified within the ESMoL toolchain. Chapter IV discusses the overall ESMoL toolchain and the steps involved in synthesizing FRODO-compatible functional code. The TrueTime extension covered in this chapter builds upon these steps. The steps reviewed in Section 2 of Chapter IV are extended with one additional step which will be detailed in later in this chapter.

Throughout this chapter we continue to make use of the example quad-rotor system to illustrate the process of creating a time-triggered TrueTime model. As reference, Figure 19 illustrates the original time-invariant Simulink controller model. It is from this starting point that the remainder of the process continues. The important blocks to keep in mind from the time-invariant model are the Plant block and the RefHandler, and OuterLoop blocks within the gum_stix block, and the InnerLoop and DataHandler blocks which reside in the robo_stix block. The RefHandler, DataHandler, OuterLoop and InnerLoop blocks are the key software components that comprise the controller and will transition into the TrueTime model. The Plant block will be directly copied into the TrueTime model and is not altered in the process.

Model Synthesis Process

The conceptual architecture of our TrueTime approach is shown in Figure 35. This figure looks very similar to that of the FRODO architecture shown in Figure 31. Again we will leverage the ESMoL model to generate both the functional code and some glue-code, but one important distinction is made between FRODO and TrueTime. FRODO implements its time-triggered scheduler directly on top of the hardware node's OS. In TrueTime there is no real OS, so instead a custom TrueTime compatible time-triggered schedule had to be implemented. This scheduler is completely reusable between ESMoL models and is responsible for implementing the time-triggered behavior ESMoL expects. TrueTime in turn builds atop the core Simulink simulation engine.

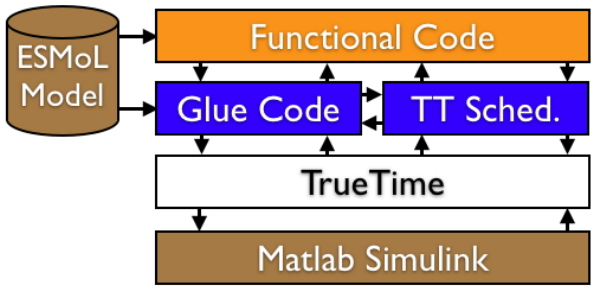

Figure 35: TrueTime Conceptual Architecture

The process of building a TrueTime model builds directly on the modeling steps discussed in Chapter IV. Two additional phases are involved in synthesizing a TrueTime model from an ESMoL model. First, a new Simulink model containing TrueTime network and kernel blocks must be generated. These kernel blocks only provide a scheduling and execution framework but do not implement task behaviors themselves. Therefore, code implementing tasks that will execute within kernel blocks must be supplied. The second phase synthesizes some "glue-code" that, when compiled with the previously generated functional code, see Chapter IV Section 2 step 3, implements the tasks the TrueTime model will execute.

Simulink Model Synthesis

The first phase, creating the new Simulink model, is itself a two step process. Due to available Matlab APIs, it is easier to synthesize an M-file script, which in turn generates a Simulink model, than it is to generate a Simulink model directly. Once generated, the M-file script is run and a new Simulink model is created with the appropriate configuration of blocks. A one-to-one correspondence exists between ESMoL nodes and buses and TrueTime kernels and networks respectively. The original Simulink model's plant and reference signal blocks must also be part of the new model. Figure 36 shows the resulting Simulink model generated from the M-file script.

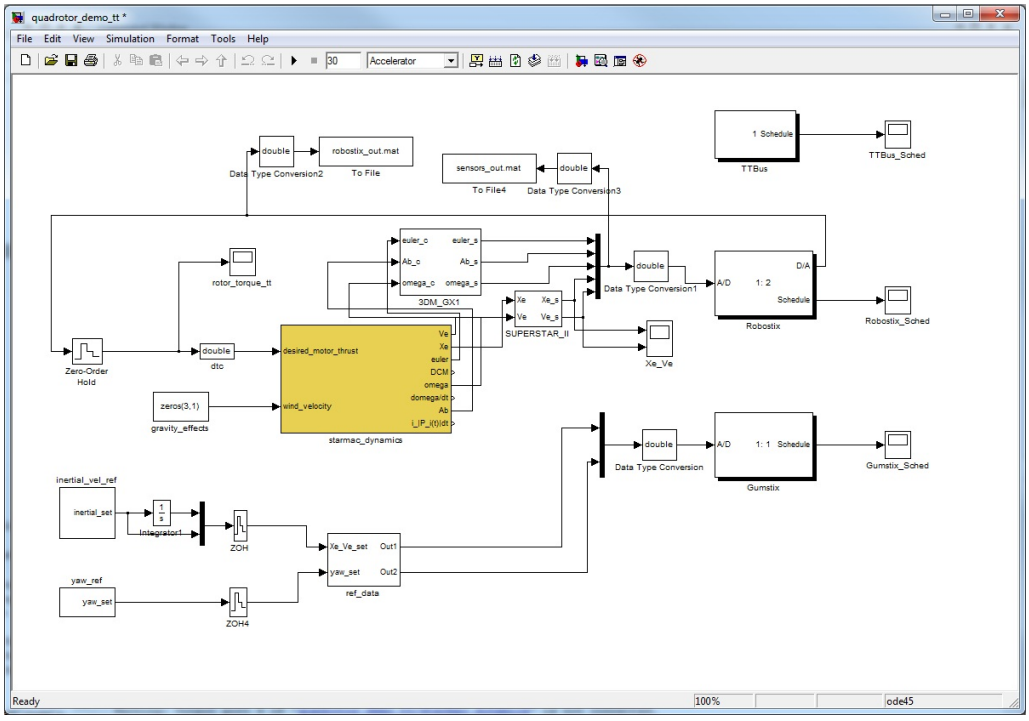

Figure 36: Synthesized TrueTime Model of the Quad-Rotor System

In our example, analog inputs are generated by the Starmac dynamics model via a simulated serial block and outputs are sent back to the Starmac block again using a serial protocol. The number and ordering of analog signals must be derived from the sensor and actuator messages defined in the ESMoL deployment configuration. Other kernel parameters such as the node number (for network identification) and initial local clock values are similarly derived from the ESMoL model. For the quad-rotor example, a network block with CSMA/CD network type is generated and its properties

are configured according to the bus information in the ESMoL model. A CSMA/CD network was chosen since our scheduler, see next section, implements the time-based bus access. While the TrueTime network block must be configured, it does not require any additional code for proper execution.

Glue-Code Synthesis

The second phase of creating a TrueTime model is to synthesized the layer of glue code that binds the functional code to the TrueTime run-time. TrueTime is able to compile and link with either M-file or C/C++ code for task implementations. In ESMoL both representations of tasks are available, M-file from the original imported controller model, and C-code from the synthesized functional code. We have chosen to leverage the C functional code since it is identical to the code that will eventually be utilized in a fully deployed control system application.

It is the glue code that implements the semantics of a time-triggered architecture on top of the TrueTime primitives. TrueTime kernels support both periodic and sporadic execution of tasks. Neither of these provide the exact timing semantics desired for time-triggered semantics as they are deadline or period driven and are not guaranteed to begin execution at a specific time. Given a static TT execution schedule, tasks should begin execution at their scheduled times. This requirement necessitates a custom execution scheduler be built on top of the TrueTime scheduler. This is analogous to implementing a TT virtual machine on top of a host OS, vis-a-vis FRODO as discussed in Chapter IV and [17].

The scheduler used within an ESMoL time-triggered TrueTime model is shown in Figure 37. Our scheduler is implemented as a high-priority periodic task and is scheduled for execution at the beginning of every hyperperiod. TrueTime structures task execution into "segments". A TrueTime task can execute in one or more segments, each of which consumes some finite amount of simulation-clock time. At the end of each segment the task returns control to the TrueTime scheduler and informs TrueTime of its execution duration via a returned double value. Any data that must persist between segments or executions must be stored and retrieved in "UserData" via TrueTime access

```
double scheduler_exec( int seg, void *data ) {
    // Get the kernel data structure
    KernelData *kernelData = (KernelData*)ttGetUserData();

    // See if we are in the start of a hyperperiod ...
    if ( seg == 1 ) {
        // Determine start of current hyperperiod
        kernelData->hyperperiodStart = ttCurrentTime();
    }
    // Otherwise we should schedule a task
    else {
        // We are woken up, now schedule the task
        ttCreateJob( kernelData->currentTask->second.c_str() );
        // Move on to the next task
        kernelData->currentTask++;
        // Double check for end of hyperperiod
        if ( kernelData->currentTask == kernelData->tasks.end() ) {
            // Reset the task list pointer
            kernelData->currentTask = kernelData->tasks.begin();
            // Increment the hyperperiod count
            kernelData->hyperperiod++;
            // And we are out of here
            return FINISHED;
        }
    }
    // Determine time of the next task to be executed
    double taskTime = kernelData->currentTask->first +
                      kernelData->hyperperiodStart;
    // Sleep until that time
    ttSleepUntil( taskTime );
    // Micro step in time
    return 0.001;
}
```

Figure 37: Online Time-Triggered Scheduler Embedded within each TrueTime Kernel

functions.

Segment 1 in our scheduler corresponds to the first segment of each execution, and thus the beginning of each hyperperiod. Our scheduler maintains a sorted map of start times to ESMoL tasks and a pointer that points to the next task to be executed according to this schedule. During segment 1, the first ESMoL task for the hyperperiod is found and its absolute start time calculated. The scheduler then sleeps until this time is reached.

When the time has arrived for an ESMoL task to be executed, our scheduler should be awakened from sleep by TrueTime. This corresponds to any segment greater than 1. ESMoL tasks are implemented as sporadic TrueTime tasks that have a priority lower than our scheduler task. Our scheduler executes an ESMoL task by scheduling it for execution in TrueTime using the *ttCreateJob*() function. Since TrueTime is set to use a priority-based scheduling scheme, and no other tasks besides our scheduler are active, as soon as our scheduler ends its segment this new job will execute. This approach ensures that an ESMoL task starts execution at its statically scheduled time. ESMoL

tasks interact with the TrueTime runtime using segments also.

```
//Code to handle the execution of InnerLoop_task
double InnerLoop_code( int seg, void *data ) {
    // Get the kernel data
    KernelData *kernelData = (KernelData*)ttGetUserData();
    // Go through the two possible phases
    switch ( seg ) {
        // Execute component code
        case 1:
            // Execute the task component
            InnerLoop_main(
                kernelData->InnerLoop_ctx,
                kernelData->OuterLoop_ang_ref->InnerLoop_ang_ref,
                kernelData->DataHandling->DataHandling_angle_data,
                &kernelData->InnerLoop_thrust->InnerLoop_ang_err,
                &kernelData->InnerLoop_thrust->InnerLoop_Torque,
                &kernelData->InnerLoop_thrust->InnerLoop_ang_vel
                );
            // Return the WCET for the task
            return 0.001900;
        // We are done
        default:
            return FINISHED;
    }
}
```

Figure 38: TrueTime Execution of ESMoL Tasks

ESMoL task executions are always contained in a single segment. All input and output messages are implemented as generated structures contained within the user data context. The ESMoL task simply calls to the corresponding functional code method that was generated for that task, passing in input data values and pointers to output data locations. This approach for input and output messages adheres to the logical execution time semantics mentioned in Chapter II. The segment finishes by returning the expected worst-case execution time (WCET) for that task given in the ESMoL model. TrueTime will always try to let the task execution continue by calling it again with a segment value of 2, but the task will signal it has completed executing by returning the TrueTime defined value FINISHED.

When our scheduler finds no more tasks to execute in a hyperperiod, it signals TrueTime that it has completed execution by returning FINISHED. This cycle is repeated each hyperperiod until the overall simulation is halted. TrueTime provides output ports on each kernel block that chart the execution states of all tasks. Figure 39 shows one hyperperiod of execution for the RS node.

The top line in the chart is the online time-triggered scheduler while all lines below it are

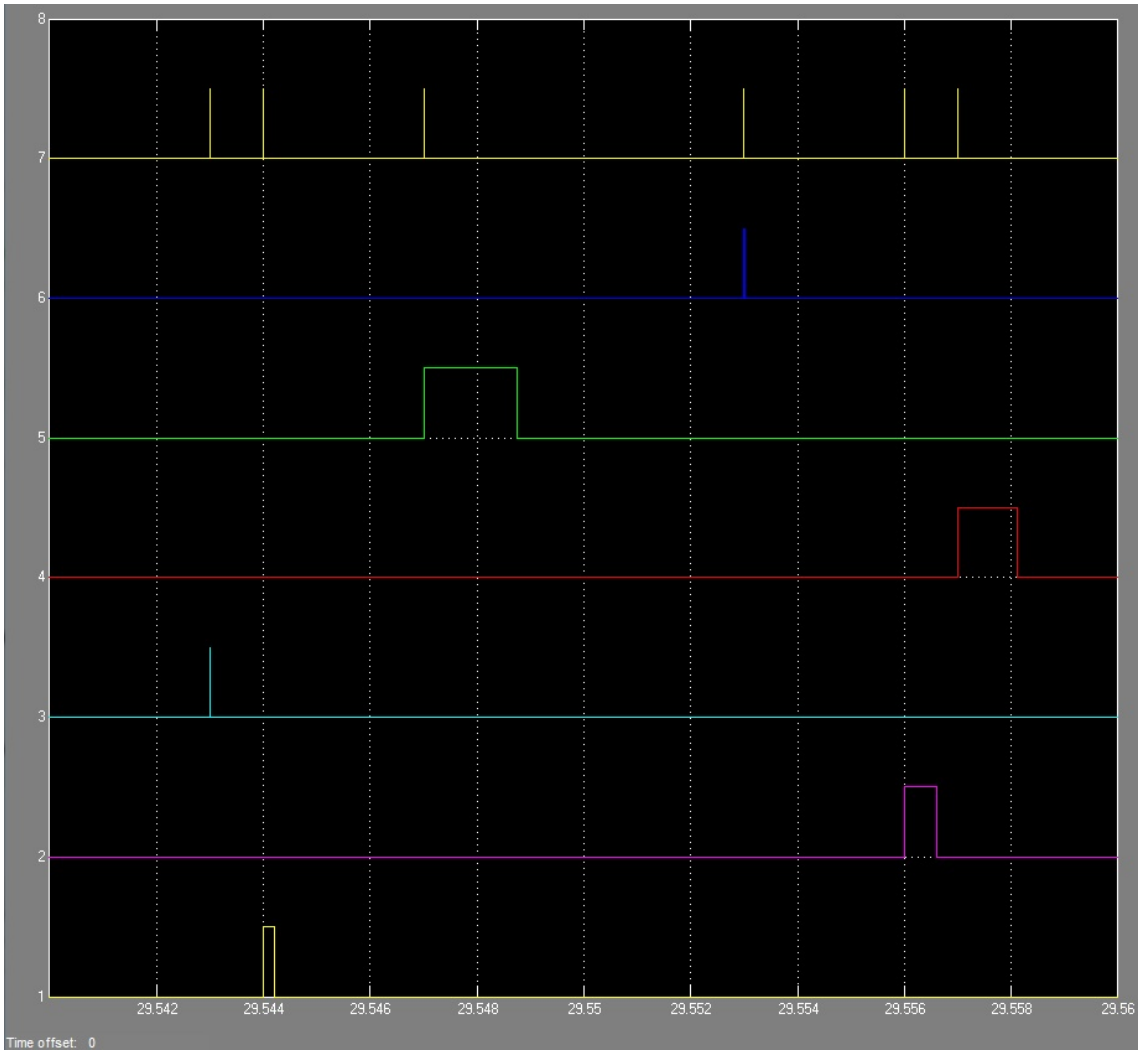

Figure 39: A single hyperperiod of the task execution schedule for the RS node

individual ESMoL tasks. For our example, there are six tasks that are executed on the robo_stix node every hyperperiod: one each for the DataHandler and InnerLoop components; one for each analog input, SerialIn, and output, SerialOut; finally, there is also a separate task for each message sent or received over the network. In this case, one message is sent from the RS node to the GS node and one received back. This ensures that all network communications remain accurate to the time-triggered execution semantics.

Experimental Evaluation

The purpose of the TrueTime model is to allow designers to simulate and analyze deployment platform induced effects on their controllers. Figure 40(a) shows the position output of the time-invariant Simulink model compared to the synthesized TrueTime model, and (b) shows the thrust commands (top) and TrueTime model error (bottom).

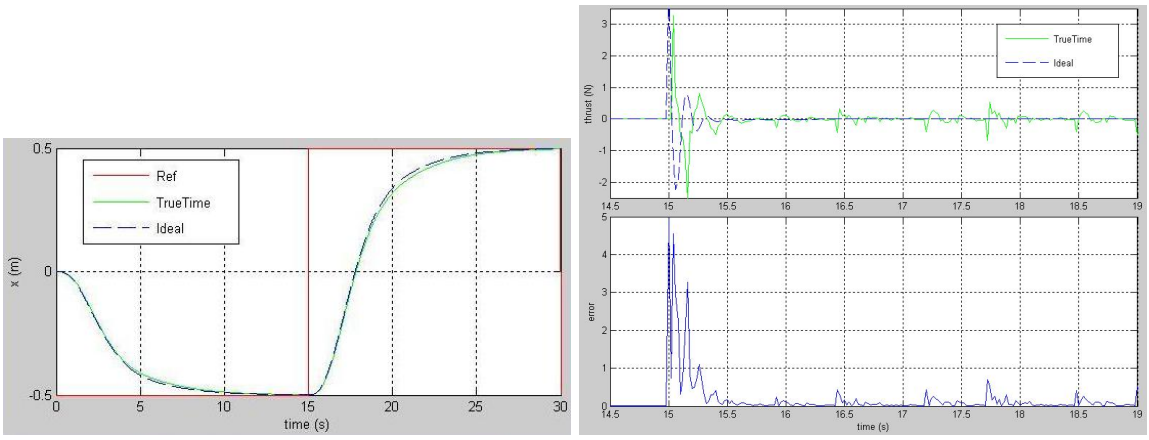

Figure 40: (a) Position Tracking (b - top) Thrust Command Comparison and (b - bottom) TrueTime Model Error

The time-invariant Simulink model does not contain any delays between the DataHandler, InnerLoop, and OuterLoop subsystems; therefore, these blocks calculate output synchronously given input from the Reference signal and the plant blocks. In contrast, the TrueTime model has propagation delays introduced by the deployment of its components onto hardware and the communications between components. The schedule causes a total of two hyperperiods to elapse between an input and its associated output response. The TrueTime model tracks position well, but does introduce nominal variance in the thrust output. From the tracking we see that this variance does not destabilize the system.

Conclusion

In this chapter, I presented an extension to the ESMoL modeling toolchain that automates the synthesis of TrueTime-based Simulink models for simulation of platform effects on deployed embedded systems. The approach is founded upon a mapping from ESMoL language elements to TrueTime

blocks and the creation of an automated interpreter that synthesizes both a TrueTime model and all necessary "glue-code" for the model. We implemented a variant of the FRODO virtual machine that runs on top of the TrueTime C++ primitives and provides the expected time-triggered execution semantics to the synthesized models. We performed multiple experiments using the synthesized TrueTime models to validate their execution correctness, assess their fidelity, and to explore possible platform effects for the experimental controllers.

CHAPTER VI

ESMOL & BIP TOOLCHAIN INTEGRATION

ESMoL models are composed of synchronous software components layered atop a time-triggered run-time execution layer. The software components are typically designed using synchronous modeling paradigms, such as Simulink, and are ultimately realized in the form of C-code. Through our definition of time-triggered systems, we require that all ESMoL components execute without side effects, store all state into an execution layer managed context, and interact with all other tasks via input and output messages. These constraints on tasks enable easier encapsulation and abstraction of behavior. Generally, ESMoL treats task behavior as opaque synchronous blocks.

The time-triggered run-time execution layer can be realized in a number of ways: via TrueTime, FRODO, or otherwise as long as the tenants of time-triggered behavior discussed in Chapter III are not violated. During nominal operation, a simplistic TT virtual machine operates in only four states: idle waiting for the next task, initiating the execution of a task, waiting for a task to complete, and a reset at the end of each hyperperiod. Each state transition is driven by the execution schedule and therefore by the node's local clock. In this sense, the VM naturally acts as a timed automata [127]. Additional complexity is added by clock synchronization and message passing, but these too can be modeled using concepts present in common timed automata formalisms.

The composition of synchronous tasks with a timed automata run-time layer results in complex system behavior that, as discussed in Chapter V, can alter the expected response as compared to a time-invariant controller. Simulation and analysis of heterogenous models is non-trivial. The BIP language and toolsuite [13, 14] are expressly designed to handle heterogeneity within a model and to provide support for simulation and analysis of such models. In this chapter, we discuss an extension to the ESMoL toolchain to support automated synthesis of models in the BIP language.

Time-Triggered Virtual Machine & Timed Automata

As a first step to creating a BIP-based representation of an ESMoL model, we discuss a formal model for networked timed automata and present the ESMoL time-triggered run-time scheduler implemented in this formalism. Using the timed-automata model of the scheduler, we can explore the system's temporal execution via simulation and validate system properties, such as deadlock freedom and node-clock synchronization bounds.

Existing literature already relates timed automata and time-triggered execution [128, 129]. A key distinction between their approach and ours is that they directly express the temporal semantics of time-triggered system within the timed automata through extensions to the modeling language itself. Alternatively, we seek to express time-triggered behavior using exclusively standard timed automata language constructs. Our approach not only allows for more complex and complete models of the TT execution layer to be created but also allows us to better leverage existing standardized timed automata tools and methods.

Timed Automata Model of Computation

In literature there are numerous alternative definitions available for timed automata and networks of TA. From [130] we adopt the following model for a network of timed automata, A:

$$A_i = \langle L_i, l_i^0, C, A, E_i, I_i \rangle \tag{9}$$

Where $A_i \in A$, is an individual TA within the network. In this tuple L_i is the finite set of locations, or states, for each timed automata. l_i^0 is the set of initial locations, with $l_i^0 \in L_i$. C is the set of clocks for the network. A is the set of actions, co-actions, and all internal τ-actions. $E \subseteq L \times A \times B(C) \times 2^C \times L$ is an edge between two locations with an action, a guard and a set of clocks that are reset when the transition is made. $I_i : L \rightarrow B(C)$ assigns invariants to locations for one timed automata in the network.

A number of existing tools support definition, simulation and verification of various types of timed automata. We chose to utilize the Uppaal tool [131] which implements the model of computation

described above. Uppaal provides methods for defining a model, simulating the execution of the model, and for analyzing system properties using computational tree logic (CTL) [132]. Uppaal models can either be created via the graphical user interface, or XML-based files defining models can be synthesized directly. Uppaal was selected due to its wide-spread adoption, mature capabilities, and its open and well documented interfaces.

A mapping from our formal model of time-triggered systems to this model of timed automata is possible. Each sequential timing set in a TT model is represented by a separate timed automata, hence the need for a network of TAs. Each of these timed automata contains at least two states: *IDLE* and *HYPERPERIOD_RESET*. As timed automata contain no concept of a cyclic executive (i.e. hyperperiod), the execution semantics of this must be emulated via transitions between these two states. A clock guard only allows transition to the HYPERPERIOD_RESET state once the TA's clock, $c_i \in C$, evaluates to at least the hyperperiod duration, $c_i \geq H$. IDLE is the state the TA inhabits at any time the system is not executing a task, computational or communication, or reseting its clock at hyperperiod end. The IDLE state is always the initial state in a model and all clocks are set at zero.

In Uppaal states are specified as *normal*, *urgent*, or *committed*. Normal states are only restricted by an invariant expression that determines if a state is valid or not. Urgent adds a restriction that no time is allowed to pass while the timed automata model is in an urgent state. Finally, committed adds an additional restriction that if a transition to a committed state is enabled it must be taken. The IDLE state is normal while the HYPERPERIOD_RESET state is committed to ensure every node resets each hyperperiod as soon as the clock equals the hyperperiod duration, $c_i = H$.

Clocks are defined similarly between our formal model of time-triggered systems and this definition of timed automata, though named as $\psi_i \in \Psi$ for TT and as $c_i \in C$ in TA. All clocks posses strictly positive real values. The clock valuation function for TT is denoted $v(\psi_i) = t$, where $t \in \mathbb{R}^+$. Similarly, $v(c_i) = t$, is the valuation function for TA clocks. The semantics of time-triggered clocks map directly to a timed automata clocks. Through the mapping from time-triggered to timed automata, each TA contains a single primary "node clock" that represents its progression through each hyperperiod. In other words, the ψ_i associated with each sequential timing set is mapped to the

node clock, c_i associated with the TA.

Tasks within a sequential timing set are represented as normal states in the TA. Task execution can be modeled in timed automata in many ways. Our mapping represents each task with one state, $EXEC_i$, and associates an "execution clock" that captures the task execution duration. Only one execution clock is needed per TA as only one task may be executing at any single instant. Transitions from the IDLE state to each task state are guarded by an expression $node\_clock \geq ST(t_i)$, where $ST(t_i)$ is the scheduled start time of the task. This transition also resets the execution clock signaling the task has begun execution. An invariant is placed on each task state, $exec\_clock \leq WCET(t_i)$, ensuring the timed automata does not spend longer than the task's worst-case execution time in the task state. A second transition connects the task execution state back to the IDLE state. A generic timed automata representing the two base states with one task state is shown in Figure 41.

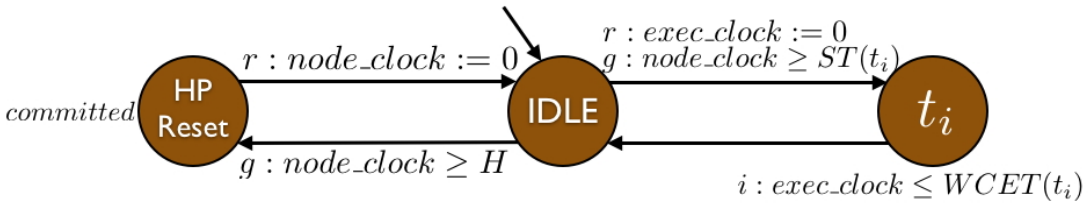

Figure 41: Generic Timed Automata of a TT Scheduler

Uppaal also supports the concept of broadcast channels. A state transition in one TA can signal an event has occurred and other timed automata can guard transitions waiting for this event signal. Channels are the mechanism by which individual timed automata are connected into a coordinated network. While broadcast channels are an extension to the formal timed automata definition given above, the concept is simple to understand.

In our mapping, the transition from IDLE to a task execution state signals the start of execution and the transition back to IDLE signals the completion of execution. An alternate approach to capturing the semantics of task execution is to create a separate timed automata for each task. These automata would have two states: IDLE and EXEC similar to the states already discussed. When the scheduler transitions to EXEC, it signals via broadcast channel the task TA to transition from IDLE to EXEC. When the task TA transitions from EXEC to IDLE, it signals, again via

broadcast channel, the scheduler TA. Representing the execution of a sequential timing set with this approach more closely mirrors the threading model used in FRODO but does not alter the semantics of the model. For simplicity we have chosen to collapse the timing set into a single TA, as shown in Figure 41.

Parallel timing sets in a timed-triggered model represent synchronization, via message passing, between multiple nodes in a system. Synchronization using broadcast channels between timed automata in a network of TA similarly captures this concept. Each message instance in a TT model, $m_i \in M$, is mapped to a global broadcast channel in the TA network. In our model of time-triggered systems, tasks that are members of parallel timing sets are also typically members of sequential timing sets, see Figure 11 for such an example.

What this dual membership implies for our TA representation is that for each task in a parallel timing set there is an associated state and transition already in a timed automata due to the tasks sequential timing set membership, or $\forall\ t_i \in parallel\_timing\_set_j$, $\exists\ EXEC_i$ state $\in TA$ and a transition from $IDLE$ to $EXEC_i$ and back must be true. In the parallel timing set, the task that is originating the message signals its execution via the broadcast channel for the message, and the transmission medium and receiver guard their transitions waiting for this signal. The return transition to the IDLE state is similarly guarded. This mapping of parallel timing sets ensures that all timed automata involved in a message transmission react synchronously which mirrors the clock synchronization via message passing characteristic of our time-triggered system definition.

In reality clock synchronization via message passing induces some skew, as discussed in Chapter III, due to transmission jitter. It is useful to include this property in the timed automata model in order to validate proper temporal execution of the models in regard to clock synchronization bounds. Only a slight embellishment of the above mapping of parallel timing sets is necessary to capture this behavior. First, a new integer variable, *skew*, is added to each timed automata. Initially its value is zero and is bounded, $[0, maxJitter]$, where maxJitter represents the maximum amount of clock skew a node could encounter in one hyperperiod. In the transition from $IDLE$ to $EXEC_i$ for a message passing task, a skew value is chosen, using the Uppaal *Select* trigger, from the bounded range and added to a list of all other skew values accumulated during the hyperperiod. At the end

of the hyperperiod these skew offset values are averaged using a robust scheme [26]. Algorithm 3 illustrates the averaging process employed with our Uppaal models.

Algorithm 3 Clock Skew Averaging

```
1: list<int> offsets //Filled with skew values during each hyperperiod
2: count textitcount = 0
3: int average = 0
4:
5: // Sort the list of offsets
6: Sort(offsets)
7:
8: // Discard the n highest and lowest values
9: for(int j=0; j¡n; j++)
10:     offsets.pop_front()
11:     offsets.pop_back()
12:
13: // Average the remaining offset values
14: while(offsets.size() ≠ 0)
15:     average += offsets.front()
16:     offsets.pop_front()
17:     count++
18: average = average / count
```

Each timed automata maintains a list of skew values through each hyperperiod. The averaging algorithm is executed upon each visit to the HYPERPERIOD_RESET state. The algorithm sorts the list of skew values and discards the n highest and lowest values, where n is the maximum number of faulty nodes allowed in the system. This provides fault tolerance and guarantees clock synchronization convergence of a group of nodes [26]. The remaining values are averaged and this value is used to adjust the node_clock during the hyperperiod.

In this section, we have shown that there exists a direct mapping from systems defined using our time-triggered formalism to equivalent timed automata. Using this mapping, any system defined using the ESMoL tools can have its temporal execution automatically converted into a TA model.

Example FRODO Scheduler TA

As discussed, the first step of creating BIP implementations of ESMoL models is realizing the time-triggered online scheduler as a simple timed automata. The scheduler within each node in an ESMoL

model is instantiated as a separate timed automata with its own local clock. Together these TA models act as a coordinating network and simulate task execution and message passing amongst nodes as if a single global clock were present.

While the model templating features present within Uppaal allows one generic TA to model any TT execution schedule using one state for idle and one state for the execution of all tasks, we instead synthesize a TA with one idle state and one individual state for each task. This is done both for simplicity and clarity. This does result in models with a larger number of visible states, but does not alter the state-space at all as the Uppaal simulation engine simply instantiates the same number of states via the template.

To illustrate the models resulting out of the mapping from time-triggered to timed automata, we return to the simple ESMoL system of two nodes first described in Chapter III and shown in Figure 11. As a quick reminder, the Node 1 calculates some value, sends it via the synchronous bus to Node 2. Node 2 then performs some calculation using that data, produces an output value and sends it back to Node 1 via the bus. Finally, Node 1 performs a final calculation and the hyperperiod concludes. The execution schedule for this system is given in Table 1.

Figures 42, 43, and 44 illustrate the timed automata models for the temporal execution of Node 1, the bus, and Node 2 respectively. For simplicity this example does not elaborate on the clock skew and offset averaging, but does display full synchronous execution semantics. A short discussion of each is provided.

Figure 42 shows Node 1 of the system, comprised of tasks A, B, C and D. The scheduled execution start time for these tasks is at 1ms, 3ms, 6ms and 8ms during the 10ms hyperperiod as seen in the transition guards leading to each of the task states. The Hyperperiod_Reset state is a committed state that is transitioned to as soon as the node_clock reaches the hyperperiod duration value.

Figure 43 shows the timed automata model for the bus in our example with tasks E and F. As per the execution schedule, they run at 3ms and 6ms respectively. The bus model also has a hyperperiod reset state.

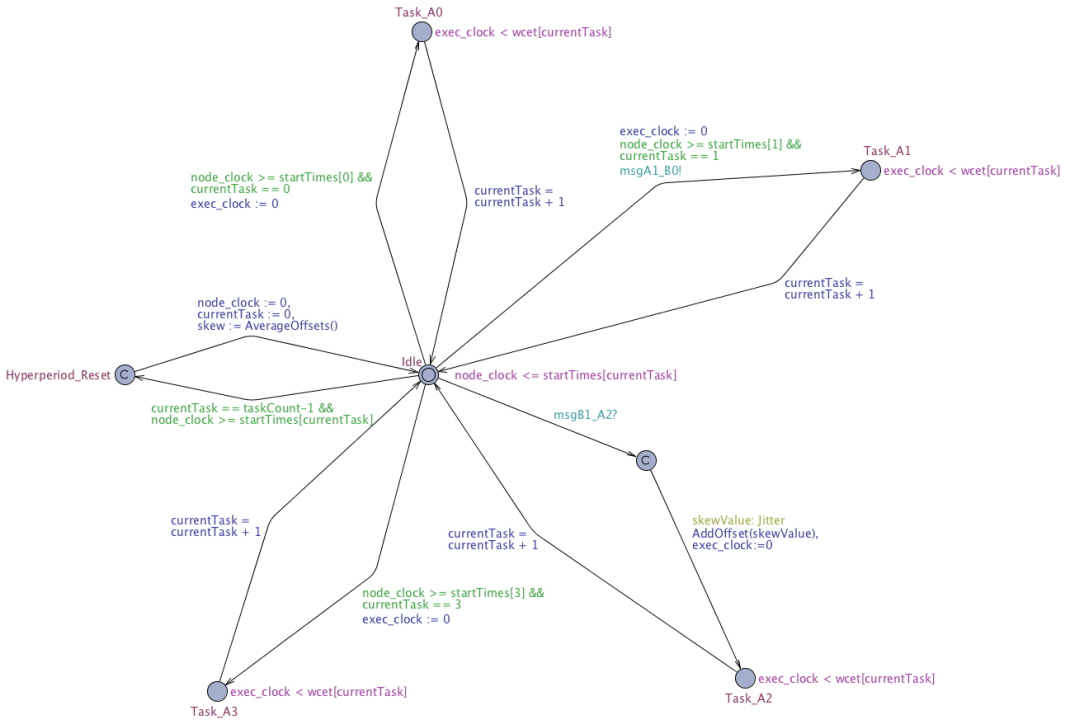

Figure 42: Node 1 Timed Automata

Finally, Figure 44 shows the model for Node 2 which is comprised of tasks G, H and I, executing at 3ms, 4ms, and 6ms respectively. Together these three models form a network which has equivalent temporal execution behavior to the original ESMoL model.

Using the Uppaal simulator it is possible to step through the execution of this network of timed automata. Variables, such as clock skews, are evaluated at each step and are given a range of possible values which they may assume.

Analysis of the Scheduler Timed Automata Network

Using the timed automata model of an ESMoL system's online scheduler, Uppaal can validate certain system properties using CTL specifications and state-space checking. With this approach it is possible to exhaustively search the state space of the timed automata model in order to, for example, ensure that key variables are maintained within an acceptable range or that certain sets of states are reached concurrently per the theoretical model.

The easiest CTL equation to specify is to make sure that deadlock is never possible within the

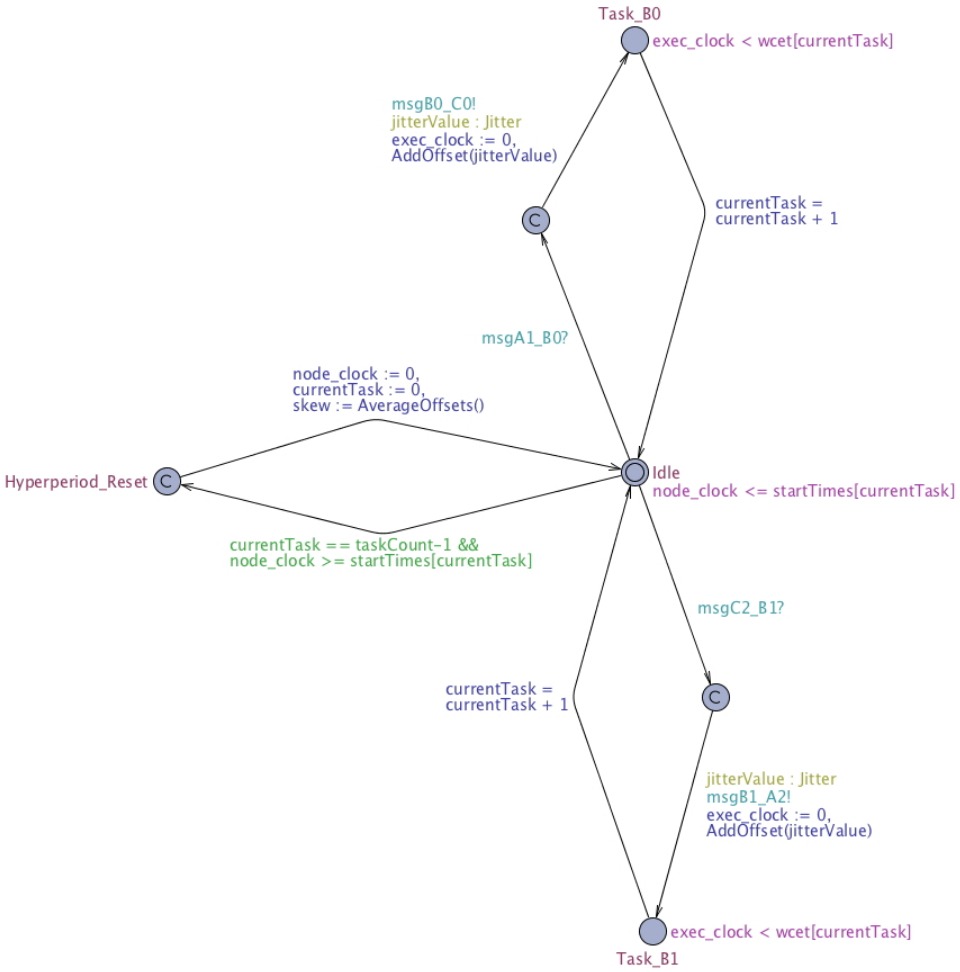

Figure 43: Bus Timed Automata

model. This is to say that there always exists at least one enabled transition. Uppaal provides a shortcut for deadlock analysis with the following equation:

$$\text{A [ ] } \textbf{not deadlock} \tag{10}$$

This equation can be read as there always exists a transition such that the "deadlock" state is never reached. A more complex analysis is to ensure that both nodes and the bus reset at the end of each hyperperiod synchronously.

$$\text{E <> Node1.Hyperperiod\_Reset } \textbf{and} \text{ Bus.Hyperperiod\_Reset } \textbf{and} \text{ Node2.Hyperperiod\_Reset} \tag{11}$$

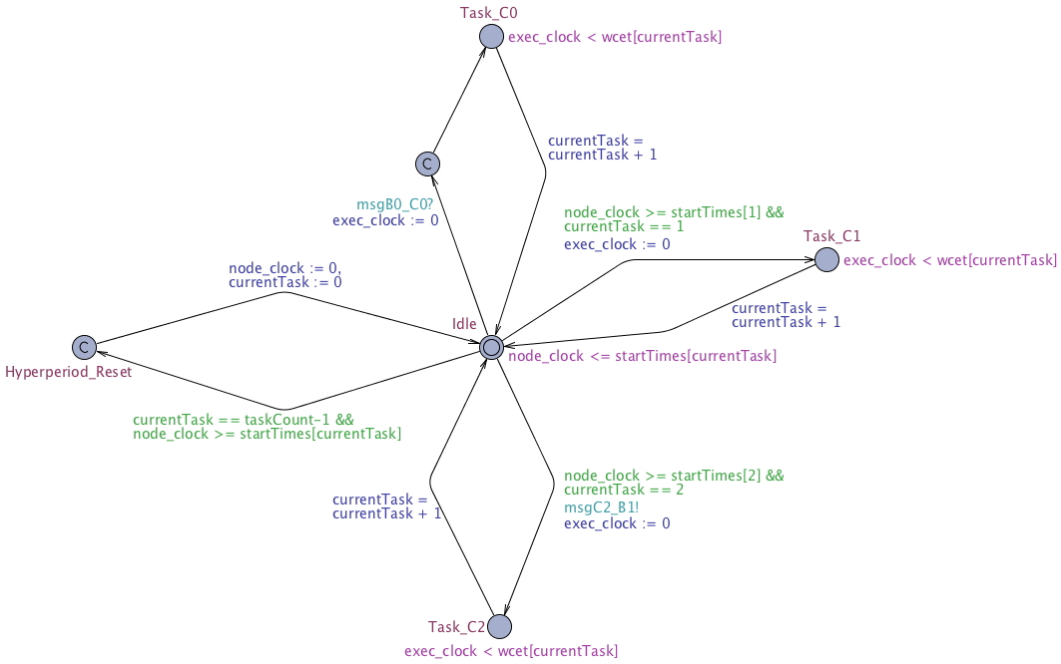

Figure 44: Node 2 Timed Automata

All three timed automata in the network must also reach the hyperperiod reset state at the same time. Parallel timing sets from the time-triggered model also translate into similar synchronous state requirements. In the following example the CTL specification requires that Node 1, the Bus, and Node 2 execute certain tasks synchronously which correspond to a message transmission.

$$\text{E <> Node1.Task\_2 } \textbf{and} \text{ Bus.Task\_1 } \textbf{and} \text{ Node2.Task\_1} \tag{12}$$

Every parallel timing set can be represented as such an equation, as the tasks in the set must react synchronously. Numerous other specifications can be derived for validating various system properties. The Uppaal tool combined with timed automata representations of ESMoL models enables meaningful analysis and simulation of the properties and temporal execution behavior of time-triggered systems.

BIP Integration

Translation of ESMoL models into timed automata allows for the analysis of certain system properties but it does not consider the results of task execution. Combining the temporal behavior of

the time-triggered scheduler with the synchronous execution of the tasks, i.e. a the heterogeneous composition of the two, is fundamental to modeling an entire system. The BIP[13, 14, 133] modeling language and toolchain supports modeling of heterogenous embedded system components. The real-time variant of its runtime engine supports timed execution of components, and is able to simulate platform effects as long as a detailed model of the platform is integrated into the overall model.

In the following section we extend our translation of ESMoL models into timed automata by further translating them into complete BIP representations. The BIP models include both the timed automata based scheduling with synchronous execution of internal tasks. This allows us to execute complete models, in real time, using the BIP runtime engine. Further, the BIP toolchain encompasses analysis and transformation tools which can prove useful to system designers, especially in the area of heterogeneous and distributed simulation.

BIP Overview

BIP stands for *Behavior*, *Interaction*, and *Priority*. Formally defined, these three layers provide the operational semantics for the language. In this subsection, a brief overview of the BIP language and its formal definition are given as a basis for understanding the transformation of time-triggered models into BIP.

From [13], the formal definition of a *Behavior* in BIP is a labeled transition system represented by the tuple: $\langle P, S, X, \rightarrow \rangle$. Where P is a set of input and output ports. S is a set of internal control states. X is a set of variables to store local data. $\rightarrow \subseteq Q \times P \times Q$ is a set of transitions each labeled by a port. Given a behavior, B, a port of this behavior, $p \in P$, is enabled iff B is in a state q and $(q, p, q') \in \rightarrow$ exists. Otherwise the port is disabled. A set of behaviors is composed using interactions and priorities.

Given a set, P, of ports from a set of behaviors, *Interactions* are defined as a non-empty subset of the ports, $a \subseteq P$. Each interaction is said to be either *enabled* or *disabled*. An interaction is enabled if all of its corresponding ports are enabled.

Priorities are defined as the following relation: $\prec \subseteq \gamma \times Q \times \gamma$, where γ is a set of interactions and Q is a set of global states. Priorities provide a partial ordering of enabled transactions.

The BIP language provides for composition via interactions, allowing for both rendezvous and broadcast style synchronization, which means that in a rendezvous interaction, all ports must be enabled in order for the interaction to be enabled, while in a broadcast only a subset of particular ports must be enabled. This provides highly flexible composition of behaviors. A detailed discussion of the BIP model of computation and its full operational semantics are found in [13].

Several recent extensions to the core BIP language have been created. In [134, 135] the BIP language and toolchain are extended to support the concept of clocks and abstract and concrete platforms. And in [136, 137] the BIP language and toolchain are extended to support automated transformations to purely send-receive style interactions and further to allow for fully distributed execution of models. It is on these extensions to BIP that we base our research. Further discussion of each extension and its relation to our work is provided below.

ESMoL to BIP Model Translation

A mapping from our formal model of time-triggered systems into BIP is possible and follows the approach developed for transforming TT systems into timed automata. Each sequential timing set in a TT model is represented by a separate BIP atomic component. Each of these atomic components contains at least one location, *IDLE*. Because of the richer semantics of BIP's state transition semantics, a separate HYPERPERIOD_RESET location is not needed. IDLE is the state the TA inhabits at any time the system is not executing a task, computational or communication, or reseting its clock at hyperperiod end. The IDLE state is always the initial state in a model.

In [134] the concept of a clock is introduced to the BIP language. This extension allows internal state transitions to be triggered by the evolution of time within a system. The BIP clock definition is practically identical to that found in both timed automata and our time-triggered model. Similar to timed automata, BIP contains no native concept of a cyclic executive. The execution semantics of this must be emulated via state transitions. When mapping a time-triggered system into BIP, each behavior is defined containing a single primary "node clock" that represents its progression through the hyperperiod. An additional "execution clock" is included for task execution. BIP initializes all clocks to zero and allows clocks to be *frozen*, *resumed* and *reset*.

Tasks within a sequential timing set are represented as locations in the atomic component. Our mapping represents each task with one state, $TASK_i$. Transitions from the IDLE location to each task execution location are guarded by an expression $node\_clockin[ST(t_i), ST(t_i)]$, where $ST(t_i)$ is the scheduled start time of the task. This transition also resets the execution clock signaling the task has begun execution. A second transition connects the task execution location back to the IDLE location. This transition is guarded based on the evolution of the execution clock relative to the task's worst-case execution time, $exec\_clockin[WCET(t_i), WCET(t_i)]$. Both the transition to the task location and the transition back are connected to exported ports in the behavior that can allow for synchronization with other components. The BIP atomic component model for the Bus in our example system is shown in Figure 45. The models for Node 1 and Node 2 would appear similar.

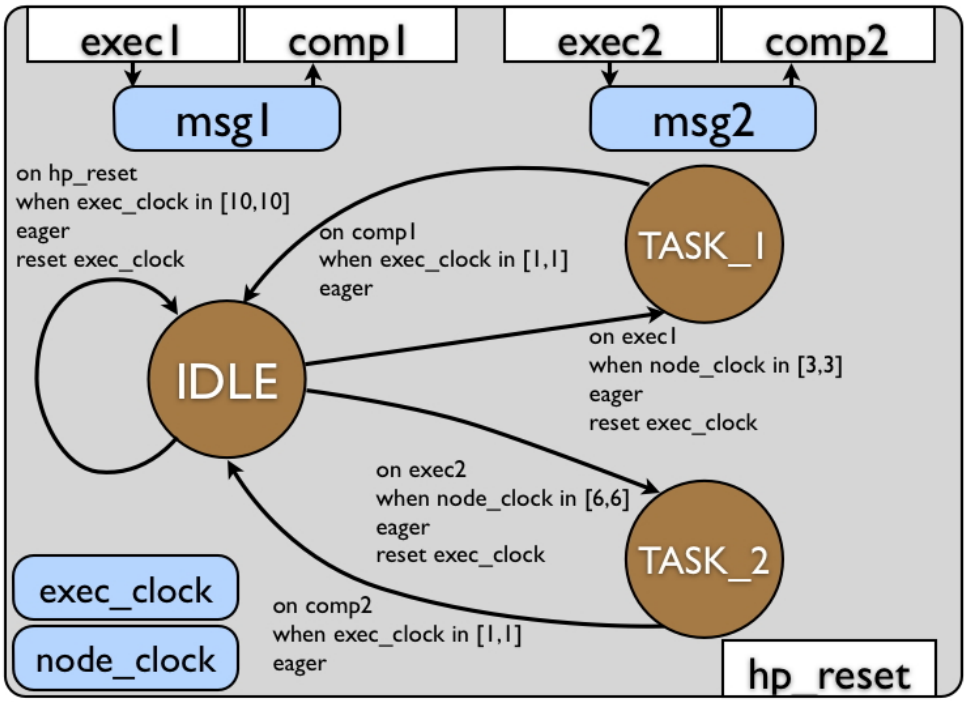

Figure 45: Detailed BIP Component for Quad-Rotor Bus

Message instances from a time-triggered model are directly translated into data structures in a BIP model. They are thus available as either input our output to executing tasks. In ESMoL, tasks are limited to synchronous execution semantics, typically derived from Simulink models, as discussed in Chapter IV. There are at least two approaches we could adopt for translating ESMoL

tasks execution behavior into BIP representations. First, as discussed in [138], a direct translation of synchronous Simulink models to BIP components is possible. Some limitations are placed on the source Simulink systems, though such translated blocks are restricted to discrete-time blocks only. ESMoL models are not bounded by these limitations so an alternate approach must be taken.

BIP allows for the execution of arbitrary C/C++ code during a state transition, though its semantics are obviously opaque to BIP. The ESMoL toolchain already supports the generation of C-code implementations for all Simulink blocks allowed in models. Additionally, ESMoL requires that all such code follow logical execution time semantics in regard to input and output messages and be side-effect free. We have chosen to leverage C-code generation capacity and integrate it into the BIP models. This approach is exactly the same as used by the FRODO virtual machine and the TrueTime simulations, and its use in the BIP models provides functional consistency across all of the ESMoL down-stream platforms.

Parallel timing sets in a timed-triggered model represent synchronization, via message passing, between multiple nodes in a system. Synchronization using interactions between atomic components in a BIP model captures this concept. For each parallel timing set in the ESMoL model, $\lambda_i \in \Lambda$, two new elements are added into the BIP model. First, a new port type is declared in the BIP model, $p_i \in P$. This port type associates a data structure with a port. The data structure mirrors that of the message associated with the tasks that are members of the timing set. Second, a new connector type is declared in the BIP model that requires the rendezvous synchronization of n ports of type p_i, where $n = |t|$, the number of task instances in the parallel timing set.

Clearly, data must move between the atomic components in order for message passing to function correctly. BIP connectors are allowed to take actions during execution. Using the task-message connectivity information contained in the time-triggered model, the connector types translated from parallel timing sets copy a message instance from the sending atomic component to the connector, and from the connector to all receiving atomic components.

Figure 46 shows the resulting BIP model for the entire example system. Node1, Node2, and Bus are each represented at atomic components and a series of connectors compose them into one system.

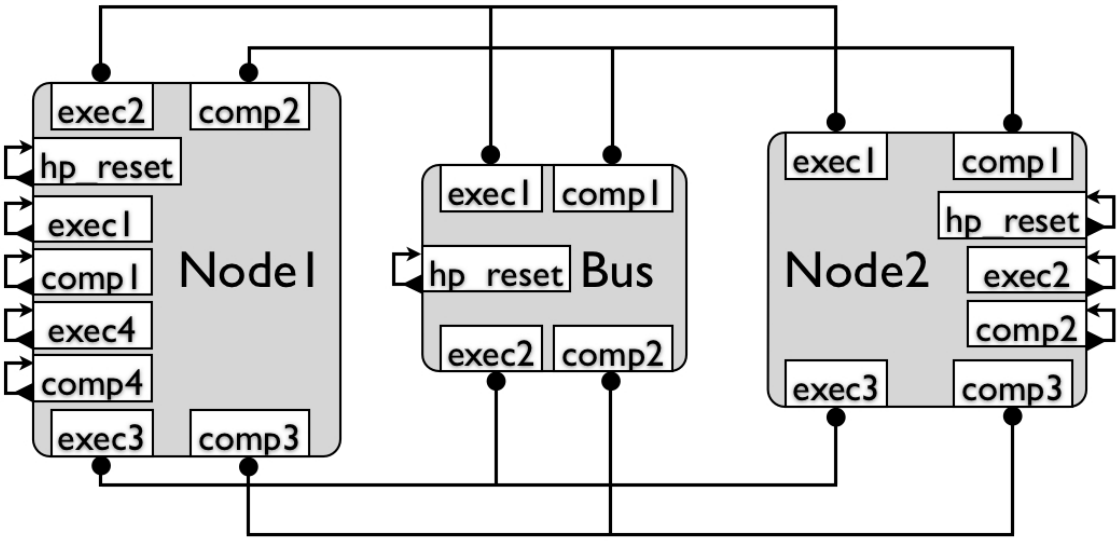

Figure 46: BIP Model of the Quad-Rotor Example System

Four triplet connectors are used to synchronize message passing between the nodes via the bus. One is responsible for the start of each transmission and one for the conclusion. Message instances are copied from the sender to the connector on transmission start and down to the receivers at the finish. Unary connectors are used for all other ports and represent the fact that these internal state transitions do not need to synchronize with external components.

The transformation of a system model from the time-triggered model of computation to the BIP MoC closely mirrors that of the transformation to timed automata. BIP allows us to create heterogeneous models that compose the clock-driven aspects of the TT scheduler with the synchronous nature of task execution. Its tools provide automatic synthesis of executable code from models, automated creation of distributed simulations involving multiple machines and running models in accordance with real-time clock constraints. All of these capabilities are powerful tools for high-confidence systems designers.

Example Execution Results

Using the approach described in the previous section, we translated the example quad-rotor system into BIP. The work included an additional translation of the plant model as well as the two nodes and the bus. Including the plant allowed simulation of the system entirely within BIP. Traces taken

from this system were identical to those taken from the TrueTime model generated from the same ESMoL model.

The results from this experiment demonstrated that a single model defined in ESMoL and anchored in our time-triggered model of computation could be mapped to several alternate models of computation, TrueTime and BIP in this case, and maintain semantic consistency. This flexibility allows embedded systems designers to utilize the best tools for a given task.

Conclusion & Future Work

Time-triggered systems are composed of two distinctly different aspects: the TT scheduler and the individual executable tasks. Approaches, such as timed automata and Simulink respectively, excel at natively modeling and simulating one or the other, but not necessarily both. Conversely, the BIP language is expressly designed to handle highly heterogeneous models. By building integration between the ESMoL and BIP languages and toolchains, this research provides a bridge by which system designers may use ESMoL's intuitive time-triggered system design tools while also having access to BIP's heterogeneous simulation and analysis tools.

At the time of this thesis's writing, two separate threads of work are evolving that are relevant to this research. As discussed above, real-time BIP augments the core language with the concept of clocks and time-constrained execution. In parallel, BIP is also being extended to support automated translation of models into fully distributed environments where individual components may execute on separate machines [136, 137]. Both of these directions align perfectly with the goals of the ESMoL project. While not yet possible, in the future these two thread will hopefully intersect so that real-time, clock-driven BIP models can be executed in a distributed environment. Once in place, this infrastructure will provide an ideal platform on which to explore large-scale time-triggered systems, and will even further reinforce the need for ESMoL-to-BIP integration.

CHAPTER VII

CONCLUSION

Summary

The process and tools for designing time-triggered high-confidence embedded systems should encompass as many useful aspects of a system's behavior as possible in order to create a more comprehensive understanding of the system. The research presented in this thesis is not intended to capture every detail of these systems, but instead aims for a subset of behaviors that are of use to system designers. This research, in conjunction with other research being conducted on ESMoL, provides a useful theoretical basis and a set of practical modeling tools for high-confidence time-triggered embedded systems.

Contributions

- **Formal Time-Triggered Model of Computation.** I have created a flexible, formal definition for time-triggered systems. The analytic properties of the model allow designers to understand the determinism, schedulability, connectivity and other properties of the design. The formal model is flexible enough to capture the design intent and execution characteristics of a wide range of the communication technologies used to connect distribute systems. From a formal system definition, it is straightforward to both synthesize system simulations and to generate executable code for realizing the actual system.

- **ESMoL Modeling Language and FRODO Time-Triggered Virtual Machine.** I have significantly contributed to the development of the ESMoL modeling language, and its associated toolchain, which is used for the design, analysis, simulation, and deployment of time-triggered high-confidence embedded systems. Designers import software components defined in external modeling tools, such as Simulink, into an ESMoL model. Details about the hardware platform are joined with the component definitions into a full description of the embedded

system. Analysis tools for time-triggered schedulability and controller stability have been integrated and can be applied against an ESMoL model. C-based functional code can be synthesized directly from the system model. I have developed the FRODO virtual machine (VM) which is the light-weight runtime layer that implements time-triggered execution and communication semantics. It is upon the FRODO VM that functional code generated from an ESMoL model executes.

- **Automated Synthesis of Time-Triggered TrueTime Models.** I have developed an extension to the ESMoL toolchain that automatically synthesizes a Simulink model for the analysis of platform effects. The TrueTime toolbox for Matlab/Simulink is a set of reusable blocks that facilitates the development of models that include the execution behavior of both hardware and software components. Building on top of TrueTime, I developed a version of the FRODO runtime that provides a time-triggered execution environment. Functional code is generated from an ESMoL system model and integrates with the TrueTime runtime code. Additionally, a new Simulink model, with the appropriate TrueTime blocks for the given hardware configuration, is automatically synthesized. This model combined with the generated code is capable of simulating possible platform effects introduced by the deployment of the time-invariant controller model onto the hardware platform.

- **Integration of the ESMoL and BIP Languages and Toolchains.** I have created a translation for models defined in the ESMoL language to the BIP language. The BIP language and toolchain are similar to ESMoL in their ability to model software components and systems. The BIP language was expressly designed with analysis of models in mind, though the toolchain also supports various methods for simulation and execution. The BIP language is able to express and compose heterogeneous models of computation, and automatically simulate them on a distributed cluster of machines. My integration of the BIP and ESMoL tool chains furthers the goal of virtual prototyping for ESMoL-defined systems through use of BIP's analysis tools and its heterogeneous and distributed runtime engine.

Acronyms

Table 6: List of Acronyms.

Acronym	Meaning	Acronym	Meaning
AADL	Architecture Analysis & Description Language	ADL	Architecture Description Language
AFDX	Avionics Full-Duplex Switched Ethernet	AFOSR	Air Force Office of Scientific Research
API	Application Programming Interface	ARINC	Aeronautical Radio, Inc.
BAG	Bandwidth Allocation Gap	BC	Basic Cycle
BIOS	Basic Input-Output System	BIP	Behavior Interaction Priority
BIU	Bus Interface Unit	CAN	Controller Area Network
CNI	Communications Network Interface	COTS	Common Off-The-Shelf
CPS	Cyber-Physical System	CPU	Central Processing Unit
CRC	Cyclic Redundancy Check	CSMA/ CD+AMP	Carrier Sense Multiple Access/Collision Detection with Arbitration on Message Priority
CTL	Computational Tree Logic	DAG	Directed Acyclic Graph
DARPA	Defense Advanced Research Projects Agency	DECOS	Dependable Embedded Components and Systems
DEVS	Discrete Event System	DIS	Distributed Interactive Simulation
DMA	Direct Memory Access	DSML	Domain Specific Modeling Language
EEPROM	Electrically Erasable Programmable Read-Only Memory	ESMoL	Embedded Systems Modeling Language
EVT	Event-Triggered Message	FAA	Federal Aviation Administration
FDMA	Frequency Division Multiple Access	FFTT	Free-Form Time-Triggered
FRODO	Original Unknown	FTG	Formalism Transformation Graph
GME	Generic Modeling Environment	GPS	Global Positioning System
HLA	High-Level Architecture	I^2C	Inter-Integrated Circuit
IMU	Inertial Measurement Unit	LCM	Least Common Multiple
LET	Logical Execution Time	LRU	Line Replaceable Unit

Table 7: List of Acronyms Cont.

Acronym	Meaning	Acronym	Meaning
MARS	Maintainable Real-Time System	MARTE	Modeling and Analysis of Real-Time and Embedded
MBSHM	Model-based Software Health Management	MEDL	Message Descriptor List
MDD	Model-Driven Development	MoC	Model of Computation
MPM	Multi-Paradigm Modeling	PDU	Protocol Data Unit
POK	Partitioned Operating Kernel	PTTM	Protected Time-Triggered Message
RAM	Random Access Memory	RT	Real Time
RTI	Runtime Infrastructure	RTL	Register Transfer Level
RTOS	Real-Time Operating System	SoC	System-on-a-Chip
SoS	System of Systems	ST	Start Time
TA	Timed Automata	TAR	Time-Advance Request
TAG	Time Advance Grant	TDL	Timing Definition Language
TDMA	Time-Division Multiple Access	TMR	Triple-Modular Redundancy
TT	Time-Triggered	TTA	Time-Triggered Architecture
TTE	Time-Triggered Ethernet	TTP/C	Time-Triggered Protocol Variant C
UART	Universal Asynchronous Receiver-Transmitter	UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol	UML	Universal Markup Language
UTTM	Unprotected Time-Triggered Message	VHDL	VHSIC Hardware Description Language
VL	Virtual Link	VM	Virtual Machine
WCET	Worst-Case Execution Time	XML	eXtensible Markup Language
ZOH	Zero Order Hold		

BIBLIOGRAPHY

[1] N. Kottenstette and J. Porter, "Digital passive attitutde and altitude control schemes for quadrotor aircraft," Vanderbilt University, Tech. Rep., 2008.

[2] E. A. Lee, "Cyber physical systems: Design challenges," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-8, Jan*, pp. 2008–8, 2008.

[3] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, and M. N. et al, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 207–220.

[4] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Springer, 1997.

[5] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE Special Issue on Modeling and Design of Embedded Software*, vol. 91, no. 1, pp. 112–126, October 2003.

[6] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 37–54.

[7] *Model-Integrated Development of Embedded Software*, vol. 91. IEEE, January 2003.

[8] C. de Vries, "Automotive: Driving embedded systems on wheels," in *ESWeek 2009*, 2009.

[9] [Online]. Available: www.mathworks.com

[10] G. Wang, "Definition and review of virtual prototyping," *Journal of Computing and Information Science in Engineering(Transactions of the ASME)*, vol. 2, no. 3, pp. 232–236, 2002.

[11] J. Eker and A. Cervin, "A matlab toolbox for real-time and control systems co-design," in *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, 1999, pp. 320–327.

[12] A. Cervin, D. Henriksson, B. Lincoln, and K. Arzen, "Jitterbug and truetime: Analysis tools for real-time control systems," in *Proceedings of the 2nd Workshop on Real-Time Tools*, 2002.

[13] A. Basu, "Component-based modeling of heterogeneous real time systems in bip," Ph.D. dissertation, VERIMAG, 2008.

[14] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in bip," in *SEFM*, vol. 6, 2006, pp. 3–12.

[15] E. A. Lee, "Computing foundations and practice for cyber-physical systems: A preliminary report," *University of California at Berkeley, Tech. Rep. UCB/EECS-2007-72*, pp. 2007–21, 2007.

[16] J. Porter, G. Karsai, P. Volgyesi, H. Nine, P. Humke, G. Hemingway, R. Thibodeaux, and J. Sztipanovits, "Towards model-based integration of tools and techniques for embedded control system design, verification, and implementation," in *Workshops and Symposia at MoDELS 2008*, ser. LNCS, vol. 5421, 2008, pp. 20–34.

[17] J. Porter, P. Volgyesi, N. Kottenstette, H. Nine, G. Karsai, and J. Sztipanovits, "An experimental model-based rapid prototyping environment for high-confidence embedded software," in *Proceedings of the 2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, vol. 0. IEEE Computer Society, 2009, pp. 3–10.

[18] R. Thibodeaux, "The specification and implementation of a model of computation," Master's thesis, Vanderbilt University, 2008.

[19] K. Tindell, H. Hansson, and A. J. Wellings, "Analysing real-time communications: Controller area network (can)," in *Proceedings 15th IEEE Real-Time Systems Symposium.* Citeseer, 1994, pp. 259–265.

[20] [Online]. Available: www.semiconductors.bosch.de/pdf/can2spec.pdf

[21] A. Benveniste, P. Caspi, P. L. Geurnic, H. Marchand, J. Talpin, and S. Tripakis, "A protocal for loosely time-triggered architectures," *Embedded Software*, pp. 252–265, 2002.

[22] A. Benveniste, "Loosely time-triggered architectures for cyber-physical systems," *Sensors*, 2010.

[23] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: The mars approach," *IEEE Micro*, vol. 9, no. 1, pp. 25–40, January 1989.

[24] H. Kopetz and G. Grünsteidl, "Ttp - a time-triggered protocol for fault-tolerant real-time systems," *Computer*, vol. 27, no. 1, pp. 14–23, 1993.

[25] TTTech, "Time-triggered protocol ttp/c high-level specification document protocol version 1.1," TTTech, Tech. Rep., 2003.

[26] J. Lundelius and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," in *Proceedings of the third annual ACM symposium on Principles of distributed computing.* ACM, 1984, p. 88.

[27] J. Rushby, "An overview of formal verification for the time-triggered architecture," in *Formal Techniques in Real-Time and Fault-Tolerant Systems.* Springer, 2002, pp. 83–105.

[28] L. Pike, "Formal verification of time-triggered systems," Ph.D. dissertation, University of Indiana, 2005.

[29] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, "The time-triggered ethernet (tte) design," *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pp. 22–33, 2005.

[30] M. Schwarz., "Implementation of a ttp/c cluster based on commercial gigabit ethernet components." Master's thesis, Vienna University of Technology, Real-Time Systems Group, 2002.

[31] K. Hoyme and K. Driscoll, "Safebus," in *Digital Avionics Systems Conference, 1992. Proceedings., IEEE/AIAA 11th*, October 1992, pp. 68 –73.

[32] A. E. E. Committee, "Arinc specification 659: Backplane data bus," Aeronautical Radio, Inc., Tech. Rep., 1993.

[33] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming." *Lecture Notes in Computer Science*, vol. 2211, pp. 166–184, 2001.

[34] ——, "Embedded control systems development with giotto," *ACM SIGPLAN Notices*, vol. 36, no. 8, pp. 64–72, 2001.

[35] T. Henzinger, C. Kirsch, M. Sanvido, W. Pree *et al.*, "From control models to real-time code using giotto," *IEEE Control Systems Magazine*, vol. 23, no. 1, pp. 50–64, 2003.

[36] W. Pree and J. Templ, "Modeling with the timing definition language (tdl)," *Model-Driven Development of Reliable Automotive Services*, pp. 133–144, 2008.

[37] E. Farcas, C. Farcas, W. Pree, and J. Templ, "Transparent distribution of real-time components based on logical execution time," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*. ACM New York, NY, USA, 2005, pp. 31–39.

[38] J. Templ, "Timing dfinition language (tdl) 1.5 specification. technical report," PreeTECH, Tech. Rep., 2009.

[39] [Online]. Available: http://www.preeTEC.com/

[40] F. Consortium, "Flexray communications system, protocol spec- flexray communications system, protocol specification, version 2.1," FlexRay Consortium, Tech. Rep., 2005.

[41] [Online]. Available: http://www.byteflight.com

[42] H. Kopetz, "A comparison of ttp/c and flexray," *Research Report*, vol. 10, p. 2001, 2001.

[43] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From simulink to scade/lustre to tta: a layered approach for distributed embedded applications," in *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*. ACM New York, NY, USA, 2003, pp. 153–162.

[44] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[45] R. Obermaisser, P. Peti, B. Huber, and C. E. Salloum, "Decos: an integrated time-triggered architecture," *e & i Elektrotechnik und Informationstechnik*, vol. 123, no. 3, pp. 83–95, 2006.

[46] H. Fuhrmann, R. von Hanxleden, J. Rennhack, and J. Koch, "Model-based system design of time-triggered architectures-avionics case study," in *2006 IEEE/AIAA 25th Digital Avionics Systems Conference*, 2006, pp. 1–12.

[47] R. DO-178B, "Software considerations in airborne systems and equipment certification," RTCA, Tech. Rep., 1992.

[48] R. DO-297, "Guidance and certification considerations for integrated modular avionics (ima)," RTCA, Tech. Rep., 2005.

[49] J. Rushby, "Partitioning in avionics architectures: Requirements, mechanisms, and assurance," 2000.

[50] A. . S. Committee, "Arinc 653," *AVIONICS APPLICATION SOFTWARE STANDARDS INTERFACE*, 2003.

[51] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a flexible real time scheduling framework," in *Proceedings of the 2004 annual ACM SIGAda international conference on Ada: The engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies.* ACM, 2004, pp. 1–8.

[52] J. Porter, G. Karsai, and J. Sztipanovits, "Towards a time-triggered schedule calculation tool to support model-based embedded software design," in *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software.* New York, NY, USA: ACM, 2009, pp. 167–176.

[53] A. Dubey, G. Karsai, R. Kereskenyi, and N. Mahadevan, "Towards a real-time component framework for software health management," Institue for Software Integrated Systems, Vanderbilt University, Tech. Rep., 2009.

[54] J. Delange, O. Gilles, J. Hugues, and L. Pautet, "Model-based engineering for the development of arinc653 architectures," *SAE International*, 2009.

[55] T. Vergnaud, B. Zalila, and J. Hugues, "Ocarina: a compiler for the aadl," *Rap. tech., École Nationale Supérieure des Télécommunications, Paris*, 2006.

[56] P. Feiler, D. Gluch, and J. Hudak, "The architecture analysis & design language (aadl): An introduction," Carnegie-Mellon University, SEI, Pittsburg, PA, Tech. Rep., 2006.

[57] J. Delange, L. P. andA. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon, "Validate, simulate, and implement arinc653 systems using the aadl," in *Proceedings of the ACM SIGAda annual international conference on Ada and related technologies.* ACM, 2009, pp. 31–44.

[58] J. Rushby, "Bus architectures for safety-critical embedded systems," in *Embedded Software.* Springer, 2001, pp. 306–323.

[59] I. Aeronautical Radio, "Arinc specification 429p1-17 mark 33 digital information transfer system (dits), part 1, functional description, electrical interface, label assignments and word formats," Aeronautical Radio, Inc., Tech. Rep., 2004.

[60] ARINC, "Aircraft data network part 7 avionics full-duplex switched ethernet network," ARINC, Tech. Rep., 2009.

[61] [Online]. Available: http://www.embvue.com/product.php

[62] S. Lee, S. W. Nam, J. Yun, K. Kim, J. Lee, J. Kim, and M. Lee, "Performance evaluation of multiplexing protocols," *SAE international congress and exposition*, 1998.

[63] G. Leen and D. Heffernan, "Ttcan: a new time-triggered controller area network," *Microprocessors and Microsystems*, vol. 26, no. 2, pp. 77–94, 2002.

[64] D. Becker, R. Singh, and S. Tell, "An engineering environment for hardware/software co-simulation," in *Proceedings of the 29th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, 1992, pp. 129–134.

[65] E. Lee and D. Messerschmitt, "Overview of the ptolemy project," *University of California, Berkeley*, 2001.

[66] J. Rowson, "Hardware/software co-simulation," in *Design Automation, 1994. 31st Conference on*, 1994, pp. 439–440.

[67] A. Hoffmann, T. Kogel, and H. Meyr, "A framework for fast hardware-software co-simulation," in *date*. Published by the IEEE Computer Society, 2001, p. 0760.

[68] Mathworks. Simulink/stateflow tools. [Online]. Available: http://www.mathworks.com

[69] D. Henriksson, A. Cervin, and K. Årzén, "Truetime: Real-time control system simulation with matlab/simulink," in *Proceedings of the Nordic MATLAB Conference, Copenhagen, Denmark*, 2003.

[70] A. Cervin, M. Ohlin, and D. Henriksson, "Simulation of networked control systems using truetime," in *Proc. 3rd International Workshop on Networked Control Systems: Tolerant to Faults*, 2007.

[71] P. V. den Bosch and E. V. de Waal, "A case study of multi-disciplinary modelling using matlab/simulink and truetime," in *Proceedings of INCOSE Symposium*, 2005.

[72] P. C. Clements, "A survey of architecture description languages," in *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*. Washington, DC, USA: IEEE Computer Society, 1996, p. 16.

[73] T. P. Consortium, "Uml profile for marte," Object Management Group, Tech. Rep., June 2008.

[74] T. Grotker, *System design with SystemC*. Kluwer Academic Publishers Norwell, MA, USA, 2002.

[75] [Online]. Available: http://ptolemy.berkeley.edu/ptolemyII/

[76] [Online]. Available: http://www-rocq.inria.fr/scicos/

[77] [Online]. Available: http://www.modelica.org/

[78] H. A. Taha, *Simulation modeling and SIMNET*, ser. Prentice-Hall International Series in Industrial and Systems Engineering. Prentice-Hall, 1988.

[79] J. Calvin, A. Dickens, B. Gaines, P. Metzger, D. Miller, and D. Owen, "The simnet virtual world architecture," *1993 IEEE ANNU VIRTUAL REALITY INT SYMP, IEEE, PISCATAWAY, NJ,(USA), 1993,*, pp. 450–455, 1993.

[80] D. C. Miller and J. A. Thorpe, "Simnet: The advent of simulator networking," *Proceedings of the IEEE*, vol. 83, no. 8, pp. 1114–1123, 1995.

[81] [Online]. Available: http://www.sisostds.org/index.php?tg=articles&topics=22&new=0&newc=0

[82] R. Hofer and M. L. Loper, "Dis today [distributed interactive simulation]," *Proceedings of the IEEE*, vol. 83, no. 8, pp. 1124–1137, 1995.

[83] P. K. Davis, "Distributed interactive simulation in the evolution of dod warfare modeling and simulation," *Proceedings of the IEEE*, vol. 83, no. 8, 1995.

[84] S. Symington, "Ieee standard for modeling and simulation high level architecture (hla) - framework and rules," *IEEE Std. 1516-2000*, pp. i –22, 2000.

[85] F. Kuhl, R. Weatherly, and J. Dahmann, *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1999.

[86] R. L. Wittman and A. J. Courtemanche. (2002) The onesaf product line architecture: An overview of the products and process. Citeseer.

[87] F. Zhang and B. Huang, "Hla-based network simulation for interactive communication system," in *Modelling & Simulation, 2007. AMS'07. First Asia International Conference on*, 2007, pp. 177–180.

[88] J. Henriksen, "Slx: The x is for extensibility," in *Proceedings of the 32nd conference on Winter simulation*. Society for Computer Simulation International, 2000, p. 190.

[89] U. Klein, S. Straßburger, and J. Beikirch, "Distributed simulation with javagpss based on the high level architecture," *SIMULATION SERIES*, vol. 30, pp. 85–90, 1998.

[90] B. Zeigler, G. Ball, H. Cho, J. Lee, and H. Sarjoughian, "Implementation of the devs formalism over the hla/rti: Problems and solutions," in *Spring Simulation Interoperability Workshop*, 1999.

[91] H. Sarjoughian and B. Zeigler, "Devs and hla: Complementary paradigms for modeling & simulation?" *TRANSACTIONS-SOCIETY FOR MODELING AND SIMULATION INTERNATIONAL*, vol. 17, no. 4, pp. 187–197, 2000.

[92] G. Zacharewicz, C. Frydman, and N. Giambiasi, "Mapping piovra in gdevs/hla environment," in *Proceedings of the 2007 summer computer simulation conference*. Society for Computer Simulation International, 2007, pp. 1086–1093.

[93] [Online]. Available: http://www.mathworks.com/products/connections/product_main.html?prod_id=696&prod_name=HLA%20Toolbox

[94] [Online]. Available: http://www.forwardsim.com

[95] [Online]. Available: http://www.mak.com

[96] T. Lu, C. Lee, W. Hsia, and M. Lin, "Supporting large-scale distributed simulation using hla," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 10, no. 3, p. 294, 2000.

[97] H. Vangheluwe, J. D. Lara, and P. Mosterman, "An introduction to multi-paradigm modelling and simulation," in *Proc. AIS2002. Pp*, 2002, pp. 9–20.

[98] J. de Lara, H. Vangheluwe, and M. Alfonseca, "Meta-modelling and graph grammars for multi-paradigm modelling in atom 3," *Software and Systems Modeling*, vol. 3, no. 3, pp. 194–209, 2004.

[99] V. Vittorini, M. Iacono, N. Mazzocca, and G. Franceschinis, "The osmosys approach to multi-formalism modeling of systems," *Software and Systems Modeling*, vol. 3, no. 1, pp. 68–81, 2004.

[100] H. Vangheluwe, "Devs as a common denominator for multi-formalism hybrid systems modelling," in *IEEE International Symposium on Computer-Aided Control System Design*, vol. 134. Citeseer, 2000.

[101] A. Bakshi, V. Prasanna, and A. Ledeczi, "Milan: A model based integrated simulation framework for design of embedded systems," *Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems*, p. 93, 2001.

[102] P. Benjamin, K. Akella, and A. Verma, "Using ontologies for simulation integration," in *Proceedings of the 39th conference on Winter simulation: 40 years! The best is yet to come.* IEEE Press, 2007, pp. 1081–1089.

[103] J. A. Miller, G. T. Baramidze, A. P. Sheth, and P. A. Fishwick, "Investigating ontologies for simulation modeling," in *Simulation Symposium, 2004. Proceedings. 37th Annual*, 2004, pp. 55–63.

[104] Y. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, pp. 381–422, 12 1999. [Online]. Available: http://www.sciencedirect.com/science/article/B6WKJ-45FKTC5-3/2/9186246cbd7c39c1c1c40633dc2f95b6

[105] H. Kopetz, "The time-triggered model of computation," in *Proceedings of the 19th IEEE Systems Symposium (RTSS98), December 1998.* Citeseer, 1998.

[106] H. Kopetz and K. Kim, "Temporal uncertainties in interactions among real-time objects," in *Ninth Symposium on Reliable Distributed Systems, 1990. Proceedings.*, 1990, pp. 165–174.

[107] A. Avizienis, "The four-universe information system model for the study of fault-tolerance," *Digest of papers*, p. 6, 1982.

[108] C. Hoefer, "Causality and determinism: Tension, or outright conflict?" *Revista de Filosofía (Universidad Complutense)*, vol. 29, no. 2, pp. 99–115, 2004.

[109] H. Kopetz, "Temporal uncertainties in cyber-physical systems," TU Wien, Austria, Tech. Rep., 2009.

[110] ——, "Sparse time versus dense time in distributed real-time systems," in *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on.* IEEE, 1992, pp. 460–467.

[111] H. Kopetz and R. Nossal, "Temporal firewalls in large distributed real-time systems," in *Proceedings of IEEE Workshop on Future Trends in Distributed Computing*, 1997, pp. 310–315.

[112] (2011). [Online]. Available: http://www.nist.gov/el/isd/ieee/ieee1588.cfm

[113] P. Pop, P. Eles, Z. Peng, and T. Pop, "Scheduling and mapping in an incremental design methodology for distributed real-time embedded systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 8, pp. 793–811, 2004.

[114] J. Porter, "Compositional and incremental techniques for high-confidence, distributed, embedded systems modeling and analysis," Ph.D. dissertation, Vanderbilt University, March 2011.

[115] H. Kopetz, "On the fault hypothesis for a safety-critical real-time system," *Automotive Software–Connected Services in Mobile Networks*, pp. 31–42, 2006.

[116] J. Sztipanovits, G. Karsai, S. Neema, H. Nine, J. Porter, R. Thibodeaux, and P. Volgyesi, "Towards a model-based toolchain for the high-confidence design of embedded systems," *Organized by the IEEE Technical Committee on Real-Time Systems*, 2007.

[117] G. Hemingway, H. Neema, H. Nine, J. Sztipanovits, and G. Karsai, "Rapid synthesis of hla-based heterogeneous simulation: A model-based integration approach," *Simulation*, 2010.

[118] [Online]. Available: https://wiki.isis.vanderbilt.edu/hcddes/index.php/Main_Page

[119] C. Schulte, M. Lagerkvist, and G. Tack, "Gecode: Generic Constraint Development Environment," http://www.gecode.org/.

[120] K. Schild and J. Würtz, "Scheduling of time-triggered real-time systems," *Constraints*, vol. 5, no. 4, pp. 335–357, Oct. 2000.

[121] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[122] (2005). [Online]. Available: http://www.tttech.com/products/ttp/middleware/real-time-operating-system/

[123] Osek/vdx time-triggered operating system specification 1.0, version 1.0 edition, july 2001. [Online]. Available: http://www.osek-vdx.org

[124] [Online]. Available: http://www.freertos.org/

[125] A. Cervin, D. Henriksson, and M. Ohlin, *TruTime 2.0 beta - Reference Manual*, Department of Automatic Control, Lund University, January 2009.

[126] [Online]. Available: http://rt.wiki.kernel.org

[127] R. Alur and D. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[128] P. Krcal, L. Mokrushin, P. Thiagarajan, and W. Yi, "Timed vs. time-triggered automata," in *CONCUR 2004–concurrency theory: 15th international conference, London, UK, August 31-September 3, 2004: proceedings.* Springer-Verlag New York Inc, 2004, p. 340.

[129] H. Kopetz, C. El-Salloum, B. Huber, and R. Obermaisser, "Periodic finite-state machines," in *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on.* IEEE, 2007, pp. 10–20.

[130] G. Behrmann, A. David, and K. Larsen, "A tutorial on uppaal," *Formal methods for the design of real-time systems*, pp. 33–35, 2004.

[131] K. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 134–152, 1997.

[132] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," in *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e.* IEEE, 2002, pp. 414–425.

[133] A. Basu, *BIP2 User Manual*, VERIMAG, 2009.

[134] T. Abdellatif, J. Combaz, and J. Sifakis, "Model-based implementation of real-time applications," *Verimag Research Report, Tech. Rep. TR-2010-14*, pp. 2010–14, 2010.

[135] T. Abdellatif, J. Combaz, and M. Poulihe, "Open real-time systems: From modeling to implementation," Verimag, Tech. Rep. TR-2011-2, 2011.

[136] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis, "From high-level component-based models to distributed implementations," in *ACM International Conference on Embedded Software, EMSOFT (to appear 2010)*, 2010.

[137] B. Bonakdarpour, J. Quilbeuf, M. Bozga, J. Sifakis, and M. Jaber, "Automated conflict-free distributed implementation of component-based models," in *Industrial Embedded Systems (SIES), 2010 International Symposium on Embedded Systems*. IEEE, 2010, pp. 108–117.

[138] V. Sfyrla, G. Tsiligiannis, I. Safaka, M. Bozga, and J. Sifakis, "Compositional translation of simulink models into synchronous bip," in *Industrial Embedded Systems (SIES), 2010 International Symposium on*. IEEE, 2010, pp. 217–220.