INTEGRATED TIMING ANALYSIS AND VERIFICATION OF COMPONENT-BASED

DISTRIBUTED REAL-TIME SYSTEMS

By

Pranav Srinivas Kumar

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

December, 2016

Nashville, Tennessee

Approved:

Dr. Gabor Karsai, Ph.D.

Dr. Xenofon D. Koutsoukos, Ph.D.

Dr. Gautam Biswas, Ph.D.

Dr. Akos Ledeczi, Ph.D.

Dr. Bharat Bhuva, Ph.D.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The decisive role of optimized and robust software in safety and mission-critical distributed real-time embedded (DRE) systems is becoming increasingly recognized. Embedded software is pertinent in a variety of heterogeneous domains, e.g., avionics [18], automotive systems [64], locomotives [97], and industrial control systems [98]. The volume and complexity of such software grow everyday depending on an assortment of factors, including challenging system requirements, e.g., resilience to hardware and software faults, remote deployment and repairability. Deployment, the procedure of installing or reconfiguring software processes on embedded hardware, becomes extremely difficult if obtaining access to such devices is limited. Large scale deployment of embedded software, for this reason, has become considerably more arduous – periodic peer reviews [15], numerous verification [74] [13] and certification methods [72] [87] are applied to maintain industry standards for safety, precision and reliability of embedded real-time software. Even still, software errors manifest in deployed systems; errors that can be extremely difficult to reproduce in a laboratory test environment.

There exists a long list of real-world scenarios where errors in embedded software implementations has cost millions of dollars and human life. Between 1999 and 2010, at least 2,200 Toyota vehicles sold in the United States experienced unintended cases of rapid acceleration causing nearly 900 accidents and over 100 deaths [22]. In 2010, Toyota recalled some 10 million vehicles, an extraordinary number given that the company sold only about 7 million vehicles during that period. Toyota engineers described the problem as a disconnect in the vehicle's complex anti-lock brake system (ABS) that caused less than a one-second lag in its operation. With this delay, a vehicle going 60 mph will have traveled nearly another 90 feet before the brakes begin to take hold. Brakes in Toyota hybrids such

as the Prius operate differently from brakes in most cars. In addition to the standard brakes, which use friction from pads pressed against rotors, the embedded software driving the electric motors help slow the vehicle. This process also generates electricity to recharge the batteries. This is a prime example of how timing errors in consumer-focused embedded software, spanning millions of lines of code, can have disastrous effects to everyday life. The Prius is Toyota's third best-selling model in the United States. The automaker recalled 2.3 million vehicles on January, 2011 because of problems with sticking gas pedals and later halted the sale of the eight models involved in the recall. Toyota's U.S. sales plunged 16 percent in January as a result, even as sales of other automakers rose.

To mitigate such software complexity, model-driven component-based software engineering (CBSE) [9, 14, 20, 37, 82] has become an accepted practice. CBSE tackles escalated demands with respect to requirements engineering, high-level design, error detection, tool integration, verification and maintenance. The widespread use of component technologies in the market has made CBSE a focused field of research in the academic sectors. Applications are built by assembling together small, tested component building blocks that implement a set of services. These building blocks are typically built from design models and captured in UML [12] class diagrams, or imported from other projects/vendors and *connected* together via exposed interfaces, providing a "black box" approach to software construction. This approach also treats software verification in a more modular fashion; the various software components can be verified individually and then composed together to derive a functional system.

Remote embedded devices, e.g., fractionated spacecraft [1], following mission timetables and hosting distributed software applications expose several concerns including strict timing requirements, complexity in deployment, repairability and resilience to faults, including mechanical failures like surface fractures, electrical failures, such as single-event upsets, and manufacturing defects, and lastly software failures such as design defects and

---

[1]A fractionated spacecraft is a satellite architecture where the functional capabilities of a conventional monolithic spacecraft are distributed across multiple modules which interact through wireless network links.

run-time faults. High-security and time-critical software applications hosted on such platforms run concurrently with all of the system-level mission management and fault recovery tasks that are periodically undertaken on the distributed nodes. Once deployed, it is often difficult to obtain a reliable period of low-level access to such remote systems for runtime debugging and evaluation. These types of DRE systems, therefore, demand comprehensive design-time modeling and analysis methods to detect possible anomalies in system behavior, like the unacceptable response times in the advanced braking systems in vehicles.

With the DARPA System F6 Project, our team has designed and prototyped a full information architecture called *Distributed REal-time Managed Systems* (DREMS) [25, 29] that addresses requirements for rapid component-based application development and deployment for fractionated spacecraft. The stack of developed software includes a design-time model-driven development tool suite [26], and a component model [66] with precise execution semantics enabling robust and analyzable software designs. The minutiae of the DREMS architecture are described in Chapter IV. The formal modeling and analysis methodology presented in this dissertation focuses on applications that rely on this foundational architecture.

The principle behind design-time analysis here is to map the structural and behavioral specifications of the system under analysis into a formal domain for which analysis tools exist. The key is to use an appropriate model-based abstraction such that the mapping from one domain to another remains valid under successive refinements in system development such as code generation. The analysis must ensure that as long as the assumptions made about the system hold, the behavior of the system lies within the safe regions of operation. The results of this analysis will enable system refinement and re-design if required before actual code development.

Figure 1a shows a *spiral model* [16] of a typical industrial software/system development life cycle (SDLC). The five stages in this cycle include requirements analysis, software design, implementation, integration testing, and design evolution. Although the intricacies

3
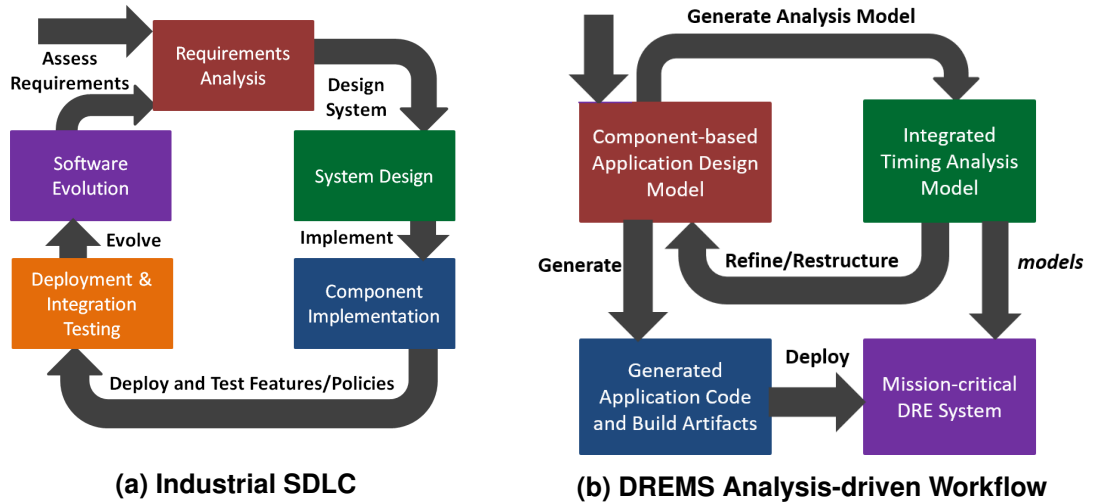
Figure 1: Embedded Software Development Lifecycle Comparison

of each stage is hidden, the large majority of industrial software development follows this life cycle. Embedded software development, especially for safety critical systems, does not lend itself well to this life cycle, mainly because the deliverable in such projects is usually not just a software package or a hardware platform but an amalgamation of both. Software development in fields like robotics is tightly coupled with the hardware; assessment of software performance is sometimes dependent on and blocked by hardware availability. Such blocking delays lead to inefficiencies in software evaluation and longer development times. It is also possible that design oversights could lead to poor timing performance, e.g., long response times to critical events, that could damage the hardware in the process. Thus, the analysis presented in this work supports and argues for a verification-driven workflow, as shown in Figure 1b. The software evaluation is performed at design-time as often as possible until the assembly is refined and optimized. Application developers use domain-specific modeling languages to structure large-scale component assemblies and modular code generation to speed up software development efforts. Moreover, domain-specific properties, such as the component wiring, patterns of interaction, such as blocking remote method invocations, asynchronous messaging etc., component execution code, and associated temporal properties, such as worst-case execution times, deadlines etc., can be easily injected

4

into such models. Using such application parameters in the *design* model, a *formal analysis model*, e.g., Colored Petri Net model (CPN), is generated. The system behavior is both simulated and analyzed by loading this model into an appropriate analysis tool, e.g., CPN Tools [75], and useful properties of the system are verified, i.e., from design-level properties like priorities and deadlines, similar timing-related properties of components such as worst-case response times are determined. By generating a bounded *state space* of the system, the execution traces exhibited by the system can be searched for property violations. Such system properties include the lack of deadlocks, deadline violations, and worst-case trigger-to-response times. The goal of this analysis is to ensure that a component-based system, an assembly of tested component building blocks, meets the temporal specifications and requirements of the system.

The results of this analysis will help improve the application enabling safe deployment of dependable components that are known to operate within system specifications. Using CBSE also enables this restructuring process as the components are not necessarily coupled software entities. So, when designing the integrated system, the analysis can be performed by assigning *time budgets* to the discrete tasks in the execution. This enables timing analysis before implementation and also uses the time budgets as requirements for efficient code implementation. These budgets are often derived from some high-level requirements and appropriately distributed between the different components in the system. The analyzed system may not necessarily be complete, but instead be in a process of evolution. As the design progresses, the system requirements become concrete and the design is re-verified at each stage to ensure the consistency of all timing guarantees.

The remainder of this dissertation is organized as follows. Chapter II describes some fundamental concepts about distributed real-time systems, component-based software and some challenges in timing analysis. Chapter III briefly describes general software testing and analysis methodologies, and summarizing related research in timing analysis and verification for distributed real-time embedded applications. Chapter IV introduces the DREMS

infrastructure and the Component Model used to experiment with and validate the timing analysis results. Chapter V discusses the Colored Petri net-based timing analysis model devised for component-based DRE systems. Chapter VI describes the scope and efficiency of the analysis methods implemented with this CPN model. Chapter VII evaluates this model with published results on analysis design, scalability and experimental validation. Finally, Chapter VIII concludes the dissertation, providing a summary of the detailed work and describing potential future work.

CHAPTER II

FUNDAMENTALS

A real-time system [55] is one where the correctness of the system behavior is dependent not only on the logical results of the computation but also on the physical time when these results are produced. Here, the system *behavior* refers to the sequence of outputs over time of the system. The flow of time is modeled as a directed line that extends from the past into the future. A slice of time in this line is called an *instant*. Any ideal (expected) occurrence at a time instant is called an *event*. An interval on this time line is called the *duration*, defined by two events, the start event and the end or terminating event. This timeline is discrete when the time line is partitioned into a sequence of equally spaced durations, called clock *ticks*. A real-time system typically changes as a function of physical time, i.e., a non-spatial continuum in which events occur in apparently irreversible succession from the past through the present to the future. If the real-time system is *distributed*, then it consists of a set of computers, *nodes*, interconnected by a real-time communication network.

Real-time systems are subject to strict operational deadlines. These deadlines constrain the amount of time permitted to elapse between a stimulus provided to the system and a response generated by the system. Consequently, the correctness of such systems depends heavily on its temporal and functional behavior. Real-time programs that are logically correct, i.e., implement the intended functions, may not operate correctly if the required timing properties are not met. Typically, such systems are classified as either soft or hard real-time systems. In soft real-time systems, missing deadlines does not completely degrade the overall system performance, e.g., delays in opening a web browser does not render the web browser useless; the browser is still functional. Hard real-time systems, however, are systems where missed deadlines could be critical, e.g., delays in pacemaker timing cycles

leading to irregular heart beats with potentially fatal consequences. It is important that any error within the system, e.g., data loss or corruption, node failure etc., be detected within a short time with a very high probability and mitigation be taken under strict timing constraints. The required error-detection latency must often be in the same order of magnitude as the sampling period of the fastest critical control loop. Then, it is possible to perform corrective action, or bring the system to a safe state. This makes the design of hard real-time systems different from soft real-time systems. The demanding response time requirements, often in milliseconds or less, preclude human intervention during normal operation. A hard real-time system must be highly autonomous to maintain safe operation. In contrast, the response time requirements of soft real-time systems is often in the order of seconds.

Timing and schedulability analysis in real-time systems usually assumes an ideally functioning software program where every step of computation performs as expected, and characterizes these steps with timing properties, such as worst-case execution times (WCET) [92] or response times [44]. Once a *timing model* of the system is developed, the behavior can be analyzed by using either a discrete event simulator, prototypical testing or formal analysis methods. This thesis concentrates on a formal analysis approach to analyzing the temporal behavior of a class of distributed real-time embedded systems.

Verification establishes a consistency between the formal system specification and the system requirements, and also between the specification and the implementation, while *validation* is concerned with the consistency between the model of the user's intentions and the system requirements. The missing link between verification and validation is the relationship between the user's expectations and the final system as built, i.e., validation can be done on multiple artifacts but ultimately the system should meet the client's expectations. Discrepancies between these notations are called *system specification errors*. Verification can usually be reduced to a mathematical analysis process, while validation must examine the system's behavior in the real-world. If properties of a system have been formally verified, it still has not been established whether the existing formal specification captures all

the aspects of the intended behavior in the user's real-world environment. To be free of specification errors, validation and specification testing are required for quality assurance. The primary verification method is *formal analysis* and the primary validation method is *testing*. The following chapter reviews various general analysis methodologies, including system-level or acceptance testing and formal verifications methods, and also selective system-level analysis methodologies for concurrent real-time systems that are related to this dissertation.

CHAPTER III

RELATED RESEARCH

3.1   Petri net-based Timing Analysis of Concurrent Systems

A Petri net [70] is an abstract, formal paradigm to model concurrent systems. The properties, concepts, and techniques of Petri nets are developed in search for natural, simple and powerful methods for describing and analyzing the flow of information and control in systems that exhibit asynchronous and concurrent activities. The major use of Petri nets has been the modeling of systems of events in which it is possible for some events to occur concurrently but there are constraints on the concurrency, precedence, or frequency of these occurrences.

Figure 2 shows a sample Petri net. The pictorial representation of a Petri net as a graph used in this illustration is common practice in Petri net research. The Petri net graph models the static properties of a system, much as a flowchart represents the static properties of a computer program.

This Petri net contains two types of nodes: circles (called *places*) and bars (called *transitions*). These graph nodes are connected using directed arcs from places to transitions and from transitions to places. If an arc is directed from node x to node y (either from a place to a transition or a transition to a place), then x is an input to y, and y is an input of x. In Figure 2, place P_2 is an input to transition t_1.

In addition to static properties, a Petri net can have dynamic properties that result from its execution. The execution of a Petri net is controlled by the position and movement of the markers (called tokens) in the Petri net. Tokens, indicated by black dots, reside in the circles representing the places on the net. A Petri net with tokens is a marked Petri net.

A transition must be enabled in order to fire; a transition is enabled when all of its input places have a token in them. Tokens are moved by the *firing* of transitions of the net. A
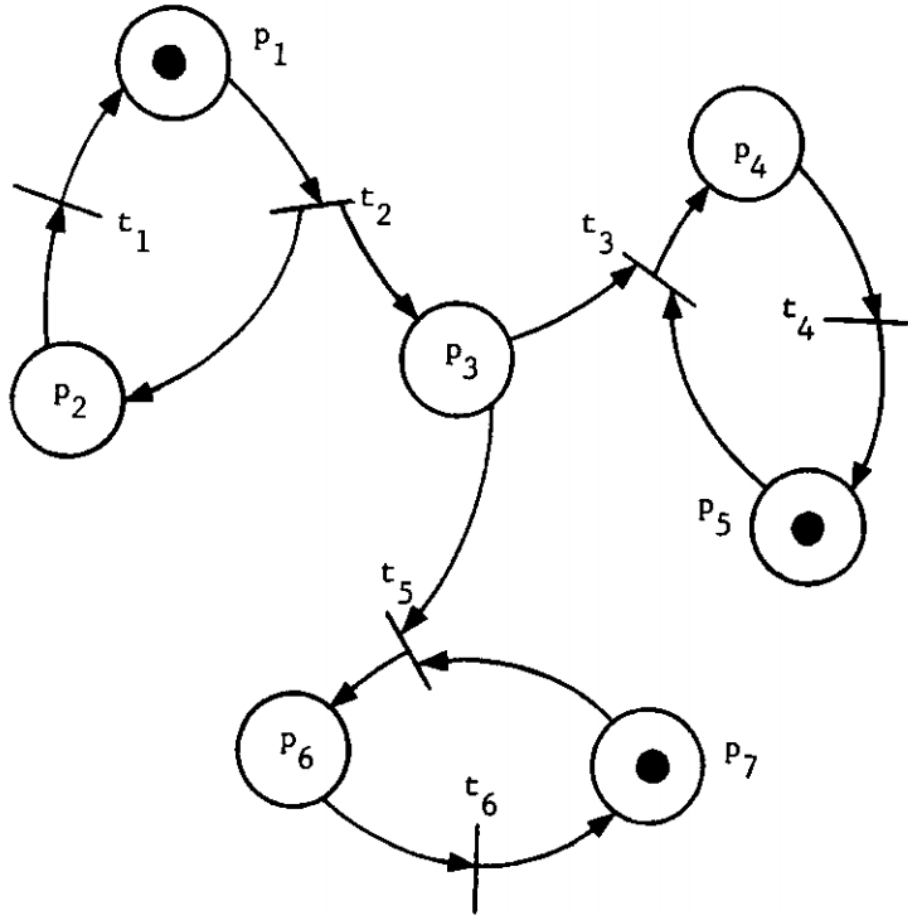
Figure 2: Sample Petri Net, reprinted from [70]

transition fires by removing the enabling tokens from its input places and generating new tokens which are deposited in the output places of the transition. In the marked Petri net in Figure 2, the transition t_2 is enabled since it has a token in its input place P_1. If t_2 fires, the token in P_1 is removed and a token is placed in places P_2, and P_3.The distribution of tokens in a marked Petri net defines the state of the net and is called a *marking*. This marking may change as a result of firing transitions. When multiple transitions are enabled for firing, one transition is chosen non-deterministically; the variability in the firing order of multiple enabled transitions gives Petri nets the capability to model non-deterministic systems, e.g., distributed systems where the ordering of events is not deterministic.

Formally, a Petri net is a five tuple (P, T, A, W, M_0), where P is a finite set of places, T

is a finite set of transitions, A is a finite set of arcs between places and transitions, W is a function assigning weights to arcs, and M_0 is some initial marking of the net. Places hold a discrete number of markings called tokens. These tokens often represent resources in the modeled system.

Petri nets enable the modeling and visualization of dynamic system behaviors that include concurrency, synchronization, and resource sharing. Theoretical results and applications concerning Petri nets are plentiful [23, 38], especially in the modeling and analysis of discrete event-driven systems. Models of such systems an be either *untimed* or *timed* models. Untimed models are those approximations where the order of the observed events are relevant to the design but the exact time instances when a state transitions is not considered. Timed models, however, study systems where its proper functioning relies on the time intervals between observed events. Petri nets and extensions have been effectively used for modeling both untimed [38] and timed systems [99]. For a detailed study of Petri nets and its applications, the reader is referred to standard textbooks [70, 76] and survey papers [63, 96, 100].

Petri nets have evolved through several generations from low-level Petri nets for control systems [76] to high-level Petri nets for modeling dynamic systems [43] to hierarchical and object-oriented Petri net structures [24] that support class hierarchies and subnet reuse. Several extensions to Petri nets exist depending on the system model and the relevant properties being studied, e.g., Timed Petri nets [89], Stochastic Petri nets [11, 57] etc. High-level Petri nets are a powerful modeling formalism for concurrent systems and have been widely accepted and integrated into many modeling tool suites for system design, analysis, and verification.

3.1.1 Example High-level Petri net Tool: CPN Tools

CPN Tools [75] is an open source tool for editing, simulating, and analysis of Colored Petri Nets (CPN) [42]. A CPN is a tuple $(\Sigma, P, T, A, N, C, G, E, IN)$ where: Σ is a finite

set of non-empty types called **color sets**. Color sets determine the types of tokens, operations, and functions that can be used in the net inscriptions (arc expressions, guards, etc.). **Places**, **transitions**, and **arcs** are defined by three finite sets P, T, and A respectively. N is a **node function** $N \subseteq (PXA)U(AXP)$. The **color function** C maps each place p into a set of possible token colors C(p), i.e., each token on p must belong to the type C(p). The **guard function** G maps each transition t into an expression of type boolean, i.e., a predicate. The **arc expression** function E maps each arc a into an expression which must be of type C(p(a)), i.e., evaluation of the arc expression must yield a resultant token that can be attached to the corresponding place. Lastly, the **initialization function** IN maps each place p into an expression which must be of type C(p), i.e., this function initializes each place with a token value of the appropriate color set type. This extends the behavior of a basic Petri net since a token represents an instance of a complex data type. In basic Petri nets, tokens in a place can represent quantity, e.g., number of tokens in a place could indicate the number of ready threads, or tokens could represent truthfulness of a state, e.g., a token in the *Blocked* place represents a globally blocked state. In CPN, tokens can not only represent a much larger variety of data structures but multiple tokens in a place can have different properties. This means that a CPN transition requires more than just the presence of a token in its input places; the tokens must have the right data (values) for the transition's guard conditions to be true, adding a layer of compactness to the representation of data.

3.2   Analyzing AADL models with Petri nets

Teams of researchers have, in the past, identified the need for integration of timing analysis methods with complex system design environments, especially in model-driven architectures [78]. Modeling languages like MARTE [65], which is based on UML, and AADL [31] (Architecture Analysis and Design Language) provide a high-level formalism to describe a DRE system, at both the functional and non-functional level. MARTE (Modeling and Analysis of Real-time Embedded Systems) defines the foundations for model-based

description of real-time and embedded systems. MARTE supports the annotation of models with information required to perform specific types of analysis, such as performance and schedulability analysis.

In general, MARTE provides a generic canvas to describe and analyze systems. The user is required to add domain/system-specific properties and artifacts on top of the generic platform. Compared to MARTE, AADL comes with a stand-alone, complete semantics that is enforced by the standard. In [78], the authors propose a bridge that translates AADL specifications of real-time systems to Petri nets for timing analysis. This formal notation is deemed to be well-suited to describe and analyze concurrent systems and provides a strong foundation for formal analysis [34] methods, such as structural analysis and model checking. The high-level goal is to check and verify AADL models for properties like deadlock-freedom and boundedness. The workflow presented here is similar to the work in this thesis in the sense that a system design model along with user-specified properties are translated into a high-level Petri net-based analysis model.

The execution behavior of the software in AADL is represented by AADL components called *Threads*. Interactions are modeled by communication places in the Petri net to trigger associated actions when AADL threads receive new data (new Petri net tokens). The thread execution is represented by an automata that has three parts: (1) thread life cycle that handles scheduling, dispatching, initialization and completion; (2) thread execution that executes thread-specific code; and (3) error management that handles potential errors. Symmetric nets [36, 93] are high-level Petri nets commonly used for analysis of causal properties in distributed systems, where tokens can carry data and nets can have a set of initial markings. Using this Petri net, the analysis uses model checking to verify (1) lack of deadlocks in the system and (2) correct causality e.g. a message sent by a producer is always received and processed by a consumer.

However, there are some potential improvements to this work. Not only is the generated Petri net structure hard to follow, it is seemingly composed of sub-Petri nets, one for each

thread (and its lifecycle) in each process. It is clear that although the transformation is sound, the generated Petri net models are going to be intractably large for complicated scenarios. The state space of the Petri net is dependent on the number of places in the net and the corresponding internal states. The generated net would not scale well for large process sets or distributed scenarios without using state space reduction techniques that rely on symmetry [83]. Such troubles can be alleviated by using a high-level Petri net such as CPN where much more information can be packed in a Petri net token. Complex token data structures reduce the number of places required to describe a system model, e.g., a list of C-style struct data structures can abstractly model a set of processors. This reduces the number of places that would be required to represent a full system. Such modeling constructs are essential in component-based systems where the full system is typically a large assembly of tested black box components. Lastly, the modeling constructs used are strictly bound to AADL concepts and cannot be easily modified for systems not modeled using AADL.

3.3   Analyzing AADL Models with Timed Petri nets

The authors in [78] have also investigated analysis of AADL models using Petri net extensions such as Timed Petri nets [77]. Using the modeling concepts and analysis capabilities of Petri net extensions means that developers can analyze for a larger set of system-level properties such as schedulability dimensioning, and deadlock detection. This work allows for efficient model-driven development and prototyping of real-time systems. Petri nets have proven to be useful mathematical means to analyze both the structure and behavior of a real-time system. Structural analysis involves analyzing a model structure to obtain knowledge about properties like circular dependencies, and causality flaws. Behavioral analysis is performed by generating and searching a bounded state space of the system, typically deducing safety properties, e.g., deadline violations. By using Timed Petri nets,

the authors insert time into any property that needs to be verified. By tagging these properties, state space queries reveal the temporal nature of system-level events that enable timing analysis results e.g. worst-case response times.

The approach presented in this work [77] uses a *Timed Petri net pattern* to model the thread life-cycle, derived from the corresponding AADL model. The state of the AADL threads are modeled using places and the life cycle is handled by the transitions. The periodicity of the thread execution is managed external to the thread pattern, by using timed tokens that represent the system clock. Multi-threaded execution is managed by the *Processor* place. The presence of a token in this place indicates an idle processor, enabling potential thread state changes.

Analysis techniques using Petri nets need to record/detects errors such as deadline violations in Petri net places. To detect missed deadlines, a deadline-detection subnet is simply added to the TPN pattern. Similar to missed deadlines, missed activations can also be detected. When the thread must be dispatched but misses its activation deadline, a detector transition fires, marking a missed activation. When model checking this system, if there is no token in some *Missed Activation* place any where in the state space of the system, then no thread activations were missed.

Similar to this work, our CPN-based analysis work uses bounded observer places [6] that observe the system behavior for property violations and prompt completion of operations. However, this work [77] only considers periodic threads in systems that are not preemptive. The non-preempt-able thread execution is evident in the need to check for missed activations. Our analysis aims to improve on this work by (1) generating a more scalable and efficient pattern-based analysis model and (2) supporting various types of hierarchical scheduling algorithms, both preemptable and non-preemptable with (3) complex periodic and aperiodic interaction patterns.

3.4 MAST: Modeling and Analysis Suite

MAST [35] is the **M**odeling and **A**nalysis **SuiT**e for real-time applications. MAST, still in development, aims to provide a set of tools that enable engineers and system integrators developing real-time applications to check the timing behavior of their system, including schedulability analysis. The techniques implemented by this tool focus on fixed-priority scheduled systems, such as the ones in commercial operating systems. The tools aims to support the timing analysis developed for both single processor [45, 54] and distributed real-time systems [68, 84].

A model describing real-time applications should not only represent the structure of the system but also the hard real-time requirements imposed on it. Most of the existing schedulability analysis methods are based on a *linear* timing and interaction model where each task is activated by the arrival of a single event or message and each message is sent by a single task. This linear model does not allow for complex interactions, e.g., nested request-response interactions and event sequences, and so in such cases the analysis methods are not applicable. The MAST model of real-time system is a rich representation. It is an event-driven model where complex dependence patterns among tasks are established, e.g., tasks may be activated by the arrival of several events at their output, making it suitable for analysis of real-time systems designed with object-oriented methodologies. This MAST model description is derived from a standard UML and MARTE description [61].

For analysis, the MAST suite includes schedulability analysis tools that use newly published research techniques such as the offset-based methods [67] that enhance analysis results, providing less pessimistic estimates than previous results [84]. The system description is specified through an textual description language that serves as the input to the analysis tools. Using UML, the real-time *view* of the DRE is described [80] by adding appropriate behavior specifying classes. The application design is linked with the real-time view to get a full description of the modeled system, along with its timing behavior and requirements.

3.5 Verification in AutoFocus 3

FOCUS is a general theory providing a model of computation based on the notion of streams and stream processing functions [17]. It is suitable to describe models of distributed, reactive systems. Based on this mathematical foundation, a tool called AutoFocus 3 [40] allows for a graphical description of systems according to this model of computation.

In AutoFocus 3, the system model is described as a set of communicating components. Each component has a defined interface (black box view) and an implementation. The interface consists of a set of communication ports. A port is either an input port or an output port, each identified by its name and its type. Components can exchange data by sending messages through output ports and receiving messages via input ports. Communication paths are called channels. A channel connects an output port to some input port, thus establishing a relationship. From the logical point of view, channels transmit messages instantaneously.

AutoFocus 3 networks are executed synchronously based on a discrete notion of time and a global clock. In this setting, a component can be either strongly causal or weakly causal. A strongly causal component has a reaction delay of at least one logical time tick which means that the current output cannot be influenced by the current input values. A weakly causal component may produce an output which depends on the current input i.e. the reaction is instantaneous. A network of strongly causal components are always well defined, e.g., unique fixed-points for recursive equations induced by channel connections always exist. Networks of weakly causal components are also well defined under the constraint that no cycles exist, i.e., no weak causal component may send a signal that feeds back to itself in the same time tick.

Input/Output automaton models are used to define stateful component behavior. The automaton consists of a set of control states, a set of data variables and a state transition function. One of the control states is defined to be the initial state of the component, while each data state variable has a defined initial value. The state transition function is defined

18

as a mapping from the current state, the current input values, and the state variable values to output values. Using this model, AutoFocus 3 supports techniques to verify the logical architecture early in the development process, e.g., automatic test case generation and model checking. Methodologies such as in [32] present the application of model checking techniques to verify the logical architecture of AutoFocus 3 models.

The formal verification process with AutoFocus 3 comprises of: (1) selecting system parts to be verified, (2) selecting requirements for the selected parts to be verified, (3) formally specifying the selected requirements, and (4) formally verifying using model checking. If the model checking succeeds, then the verification is finished, but if the model checking fails, then analysis is required to identify the reason, e.g., implementation error, requirement formalization error, etc. This analysis uses Linear Temporal Logic [33] to convert informal textual specification, e.g., "The Adaptive Cruise Control (ACC) starts by driver interaction only" to a formal temporal logic specification using boolean and temporal operators. The system model is exported into the modeling notation of the model checking tool SMV [59, 60] used by the AutoFocus 3 project.

The workflow on the above verification methodology is fairly tenuous. For any system part, a state-transition model of the part has to be constructed; this model represents the functional properties of the part. Then, all of the informal textual requirements of the system part need to be translated into LTL specifications by the user and then fed into the integrated model checker. One of the reasons the model checking could fail is improper formalization of the requirements. Secondly, the approach does not seem to model the actual execution of the software, i.e., code execution on top of layers of management software running on appropriate hardware. Thus, the method is best utilized for identifying logical errors in the system design or inconsistent property specification. The approach does not necessarily model or analyze the complete timing behavior of the component processes and is applicable mainly to early designs where the component assembly is being prepared for integration.

DESIGN MODEL: DISTRIBUTED MANAGED SYSTEMS (DREMS)

4.1 DREMS Component Model

Timing analysis of component-based software, as presented in this dissertation, is targeted towards the DREMS component model (**D**istributed **RE**altime **M**anaged **S**ystem) [29] [66]. DREMS was designed and implemented for the class of distributed real-time embedded systems that are remotely deployed and characterized by strict timing requirements, e.g., a cluster of satellites, UAV swarms, disaster relief robots, etc. DREMS is a software infrastructure for the design, implementation, deployment, and management of component-based distributed real-time embedded systems. The infrastructure includes design-time modeling tools [26] that integrate with a well-defined and fully implemented component model [51, 66] used to build component-based applications. Rapid prototyping and code generation features coupled with a modular runtime platform automate the tedious aspects of the software development and enable robust deployment and operation of mixed-criticality distributed applications. This chapter elaborates on the DREMS component model, the component operation execution semantics, and the process scheduling aspects, i.e., properties of the DREMS software stack that are relevant for generating a timing analysis model.

Figure 3 presents a typical DREMS-style component. Component-based software engineering relies on the principle of assembly, i.e., large and complicated systems can be iteratively constructed by composing small reusable component building blocks. Each *component* contains a set of communication ports, interfaces, a message queue, time-triggered event handling and state variables. Using ports, components communicate with the external world. Using interfaces and message passing schemes, components process requests from other components. This interaction mechanism lies at the heart of component-based
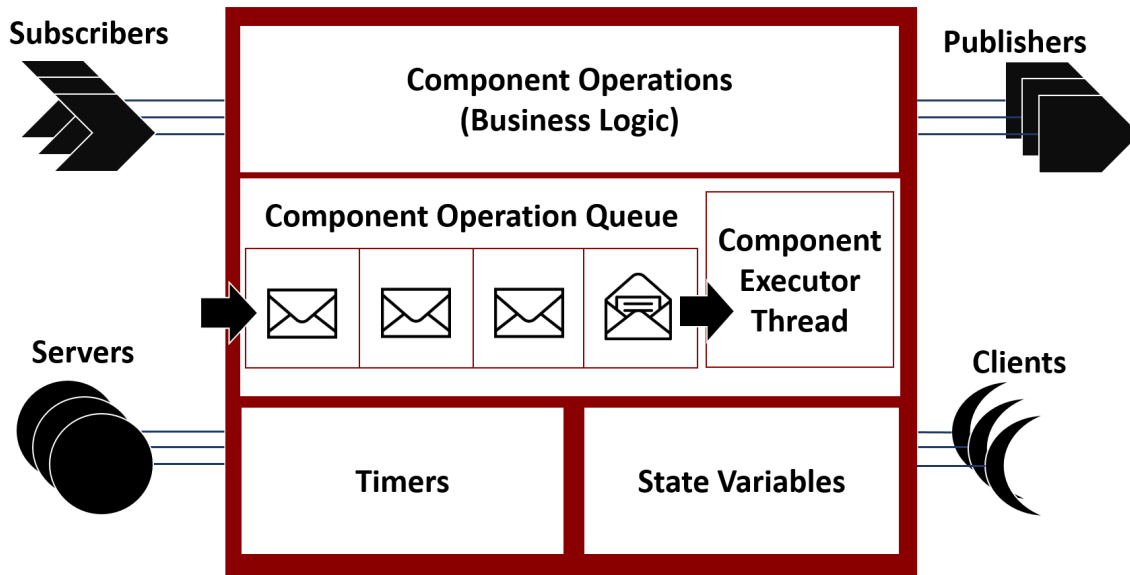
Figure 3: DREMS Component

software. Each DREMS component supports four basic types of ports for interaction with other collaborating components. These ports are Facets, Receptacles, Publishers, and Subscribers. A component's **facet** is a unique interface that it exposes to its clients. This interface can be invoked either synchronously via remote method invocation (RMI) or asynchronously via asynchronous method invocation (AMI) [73, 88]. A component's **receptacle** specifies an interface required by the component in order to function correctly. Using its receptacle, a component can invoke operations on other components using either RMI or AMI. A **publisher** port is a single point of data emission. This port emits data produced by a component operation. A **subscriber** port is a single point of data consumption, feeding received data to the associated component. Publishers and Subscribers enable the OMG DDS publish/subscribe [30] style of messaging. More details on this component model can be found in [66].

4.2 Component Execution Semantics

An *operation* is an abstraction for the different tasks undertaken by a component. These tasks are implemented by the component's source code written by the developer. Application developers provide the functional, *business-logic* code that implements operations on local state variables and inputs received on component ports. For example, a PID controller [8] could receive the measured value of a process variable from a *sensor* component, and using the relevant gains and set points (desired value for the process variable), calculate a new control signal to be sent to the *actuator* component.

In order to service interactions with the underlying framework and with other components, each component has a *message queue*. This queue holds operation requests received from another component, i.e., messages, service requests or responses, and timer activations. Each request is characterized by a priority and a deadline. Priority refers to the relative importance of one operation over another within the scope of the component. Deadline refers to the worst-case duration from the arrival of a triggering event to the completion of the response operation. If the execution of the operation takes beyond its deadline to complete, then a *deadline violation* is said to have occurred.

Each DREMS component has a separate execution thread that handles the execution of all component operations. This executor thread picks the next available request from the message queue and executes the operation to completion, i.e., the operation scheduling is non-preemptive. So, all operations in the queue, regardless of priority, need to wait for the currently executing operation to complete. Allowing only a single executor thread per component and enforcing a single-threaded non-preemptive scheduling scheme on the operations helps avoid synchronization primitives for internal state variables and establishes a more easily analyzable system. It is true that multi-threaded solutions to operation scheduling would avoid starvation, i.e., operations in the queue will not have to wait forever if the currently executing operation is blocked on a resource. We address such cases in our experimental evaluation (Section 7.5.7). However, the DREMS execution of application

components is still the more predictable one and lends itself more easily for analysis. The non-determinism in multi-threaded executions causes a tree of possible behaviors, leading to a common analysis challenge called state space explosion [53]. Keeping the operation execution to a single thread per component (1) bounds the overall number of threads in the system leading to a more tractable analysis, and (2) reduces the degree of concurrency, making the analysis computationally less demanding.
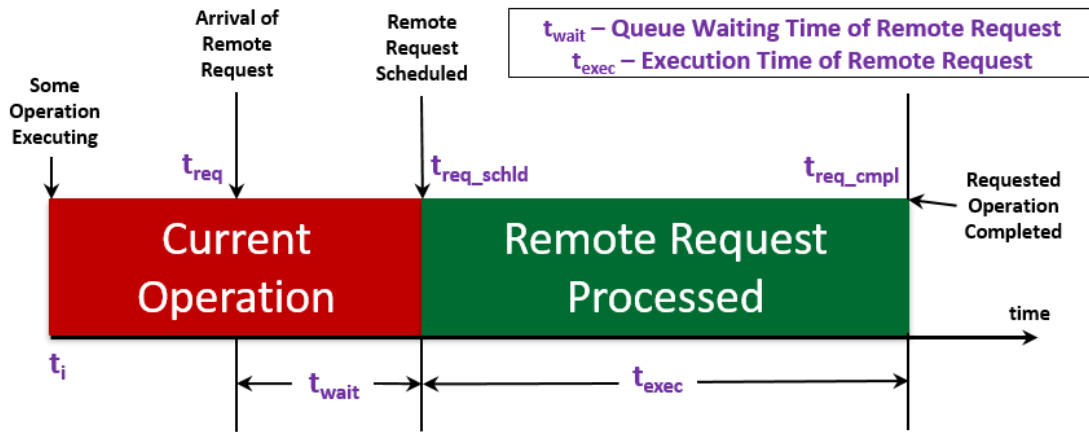
Figure 4: Component Operation Execution Semantics

Figure 4 shows the execution semantics of a component operation executed on the component's executor thread. At some time t_i, the component executor thread is busy executing an operation – component operations can be triggered into execution by the (1) expiry of a timer, (2) the arrival of a subscription message, or (3) the arrival of a service request. t_req represents the arrival time of a remote request. t_wait is the wait time of this request in the message queue while the current operation is still executing. t_req_schld is the time stamp at which the current operation completes executing. At this time, the remote request is finally scheduled for execution. t_req_cmpl is the time stamp at which the remote request completes. The overall time taken by the component to respond to this request is calculated as: $t\_wait + t\_exec = t\_req\_cmpl - t\_req$.

Simplifying assumptions include that this component is the only component thread

23

executing on the CPU assuming a single core CPU. This is a simple scenario showing how a single operation on a single component is affected by the operation scheduling semantics. When there are multiple components, the wait times of the remote request are worsened by OS scheduling non-determinism when executing the component threads.

If the executor threads of various components are of different priorities, then the highest priority ready executor thread is always chosen. If these threads are of equal priority, then round-robin scheduling is carried out, i.e., multiple components execute concurrently on one CPU. Round-robin scheduling assigns a fixed time slice to each thread, e.g., time quantum of 4 ms, in equal portions and in circular order, handling all threads without priority.

4.3   Temporal Partition Scheduler

DREMS components are grouped into processes that are assigned to temporal partitions, implemented by the DREMS OS scheduler. This scheduler was implemented by modifying the behavior of the standard Linux scheduler, introducing an ARINC-653 [7] style temporal and spatial partitioning scheme. The primary goal of this scheme is to enable isolation or partitioning of processes so as to prevent one process from adversely affecting any other process in a different partition.

Temporal partitions are periodic fixed intervals of the CPU's time. Threads associated with a partition are scheduled only when the partition is active. This enforces a temporal isolation between processes assigned to different partitions. The repeating partition windows are called *minor frames*. The aggregate of repeating minor frames is called a *major frame*. The duration of a major frame is called the *hyperperiod*, which is typically the lowest common multiple of the partition periods. Each minor frame is characterized by a period and a duration. The period specifies how often this partition becomes active and the duration defines how much of the CPU time is available for scheduling the runnable threads associated with that partition. Figure 5 shows a sample temporal partition schedule.
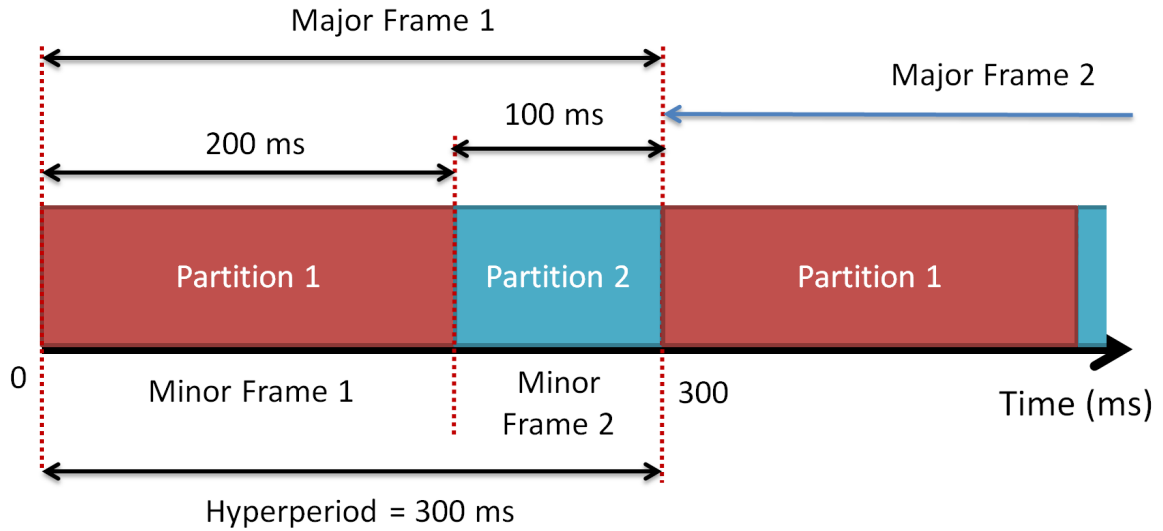
Figure 5: Sample Temporal Partition Schedule with Hyperperiod = 300 ms

By confining applications to partitions, i.e., its own memory space and temporal window of possession of computing resources, this scheduling scheme aims to enable safety and timeliness in mission-critical systems; safety by isolating processes in different applications and security levels from interacting with each other, and timeliness by providing processes with a guaranteed slice of the CPU time.

4.4 Motivation to use DREMS

DREMS supports a wide variety of interaction patterns: synchronous and asynchronous service-oriented request-response style communication, and non-blocking publish-subscribe style communication. The wide variety of interactions and communication mechanisms are inspired by other common industrial component models such as CIAO [90] and ACM [27], and the execution semantics are precisely defined and implemented. A qualitative evaluation of its capabilities [66] show that although the model was designed for fractionated spacecraft, DREMS is suitable for a variety of distributed and embedded environments. All of these properties make this component model very generic and a suitable target for the timing analysis work presented in this thesis.

CHAPTER V

COLORED PETRI NET-BASED MODELING METHODOLOGY

5.1 Problem Statement

Consider a set of mixed-criticality component-based applications that are distributed and deployed across a cluster of embedded computing nodes. Each component has a set of interfaces that it exposes to other components and to the underlying framework. Once deployed, each component functions by executing operations observed on its component message queue. Each component is associated with a single executor thread that handles these operation requests. The nature of mixed-criticality means that these executor threads are scheduled in conjunction with a known set of high-criticality system threads and other low priority *best-effort* threads. All scheduled threads are also subject to a temporally partitioned scheduling scheme.

System Assumptions:

1. Knowledge about the component definition i.e. the properties of all ports and timers in each component, e.g., priority of the operations, periodicity of the timer, operations bound to timers, servers and subscribers.

2. Knowledge about the mapping between ports/timers and component operations, i.e., how each component functionality is exposed to the component's environment.

3. Knowledge about the sequence of computation *steps* of finite duration that are executed inside each component operation. This is dependent on the operation's business-logic code written by the application developer.

4. Knowledge of worst-case time taken by the computational steps. There are some exceptions to this assumption, e.g., blocking times on RMI calls cannot be accurately judged as these times are dependent on too many external factors, e.g., the nature of

26

the process scheduling on both the client and the server, the business logic of the server operation that could include RMI calls to other remote servers etc.

5. Knowledge about the component assembly, i.e., the connections between instantiated components that form communication topologies.

6. Knowledge about the mapping between application processes and hosts, i.e., the actual embedded computer on which the process will execute.

7. Knowledge about the temporal partitioning schedule enforced by the operating system on each host.

8. Knowledge about the mapping between application processes and temporal partitions on each host.

Using these assumptions, the problem here is to ensure that the temporal behavior of the composed system meets its end-to-end timing requirements, e.g., trigger-to-response times between distant sensors and actuators. Providing this guarantee implicitly requires that communicating components in a component assembly meet individual timing deadlines. Following the DREMS component model execution, a blocking I/O operation blocks a component from attending to any other requests till the operation is completed. Such blocking interaction patterns can propagate large delays to other components, especially in a highly connected system. A useful analysis result here would not only be in identifying end-to-end timing violations but also tracing delays within individual components. Tracking timing violations enables the analysis in identifying the causes for the anomalies, e.g., nontrivial circular dependencies or scheduling delays. If an abstract model of the business logic of component operations is also encoded in the analysis model, then inefficient coding practices such as wasteful loops can also be marked as probable causes for deadline violations.

Individual components need to be analyzed to identify the *pure execution times* of the various computational steps in the component operations, i.e., the amount of time taken

by the various steps in an operation to complete if the corresponding component executor thread was the only thread executing on the system with zero CPU contention. When a set of tested components are composed together, each component's execution is affected by various factors including scheduling delays, network communication delays, blocking delays and other interaction-specific variabilities. Any timing analysis model for component-based software should account for such factors.

There are two important challenges to modeling and analyzing DRE systems: scope and abstraction level. The scope of the analysis here should be the full system of composed components. The abstraction level of the analysis must include enough detail to account for the various timing delays mentioned above while also not capturing all aspects of low-level code. A highly detailed and dynamic low-level model is necessary for simulation but not ideal for model checking and verification-based analysis due to issues like state space explosion. Also, highly composable system designs provide recombinant components that can be selected and assembled in various combinations to satisfy user requirements. In such cases, the analysis model must be efficiently capable of tackling changes in component assembly, e.g., moving components to separate processes or devices, adding or removing additional instances of the same component, etc. This is a challenge when building and non-trivially generating an analysis model from a system design. Thus, efficiency, scalability and extensibility are also modeling requirements for our timing analysis.

5.2 Colored Petri Net-based Analysis Model

Petri nets have been introduced in Section 3.1. Ordinary Petri nets have no types and are not modular. Tokens are abstract dots that represent the presence or absence of some entity. Tokens in a place could represent the presence of a message, or availability of resource, i.e., the state of execution of the place that is holding the token. The number of tokens in a place could therefore be used to represent the quantity of some available resource. Also,

ordinary Petri nets are flat structures; no hierarchy can be established to make the model more readable or concise.

With Colored Petri nets (CPN) [42], it is possible to use data types and complex data manipulations – each token has attached a typed data value, called the token *color*. It is also possible to make hierarchical descriptions, i.e., a large model can be obtained by combining a set of submodels with well-defined interfaces between submodels and well-defined semantics of the combined model. Furthermore, each submodel can be reused. One of the primary reasons for choosing Colored Petri Nets over other high-level Petri Nets such as Timed Petri Nets or other modeling paradigms like Timed Automata is because of the powerful modeling concepts made available by token colors. Each *colored token* can be a heterogeneous data structure such as a *record* that can contain an arbitrary number of fields. This enables modeling within a single *color-set* (C-style `struct`) system properties such as temporal partitioning, component interaction patterns, and even distributed deployment. The token colors can be inspected, modified, and manipulated by the occurring transitions and the arc bindings. Component properties such as thread priority, port connections and real-time requirements can be easily encoded into a single colored token, making the model considerably concise.

The CPN analysis model, as modeled in CPN Tools [75], is shown in Figure 6. *Places*, shown as ovals, in this model contain colored (typed) tokens that represent the state of interest for analysis e.g. *Clocks* place holds tokens of type *clocks* maintaining information regarding the state of the clock values and temporal partition schedule on all computing nodes. *Transitions*, shown as rectangular boxes, are responsible for executing this model, progressing the state of the modeled system and transferring tokens between places. *Arcs*, between transitions and places dictate the token flow and data structure manipulations. All arcs contain *inscriptions*, which are essentially function calls, written in Standard ML [62], that manipulate token structures e.g. arc inscriptions in the arc from the transition *Timer_Expiry* to the place *Timers*, manipulate all timer tokens by updating the timer expiry

29

offsets. When a transition fires, every input arc to this transition is evaluated first. Using the input tokens and the arc inscriptions on all output arcs, tokens are released to all output places i.e. all input arcs are associated with token values before the transition fires and output arc tokens are calculated after the transition fires. This process of associating a token value to an arc is called an *arc binding*. So, every transition firing leads to the production of strict arc bindings used to (1) evaluate guard conditions on the transitions and (2) produce output tokens.
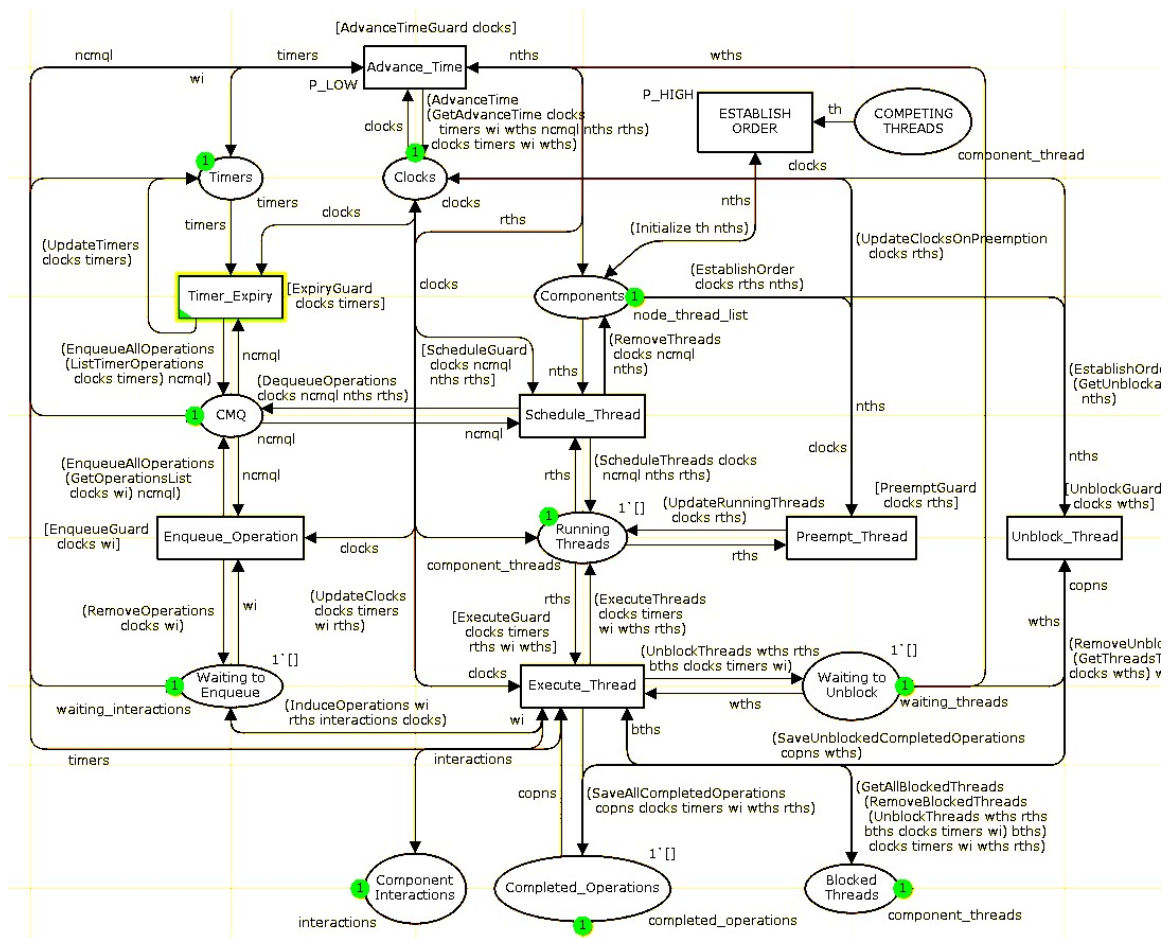
Figure 6: Colored Petri Net Analysis Model

From the design model of the system, we generate the initial CPN tokens that are injected into places in this analysis model. The modeling concepts in Figure 6 can be divided

and categorized based on system-level concepts being analyzed. Figure 7 shows the organizational structure of this CPN. Below, we describe each of these structural divisions in detail.

Figure 7: Analysis Model - Structural Aspects

5.2.1 Model of Time

Appropriate choice for temporal resolution is a necessary first step in order to model and analyze threads running on a processor. The OS scheduler enforces temporal partitioning and uses a priority-based scheme for threads active within a temporal partition. If there is no active OS-level temporal partitioning, the analysis model assumes the presence of a

single infinitely long temporal partition in which all the application processes can execute. If multiple threads have the same priority, a round-robin (RR) scheduling is used. Each thread has a *quantum* which is effectively a duration of time where the thread is allowed to keep hold of the CPU if the thread remains runnable and the scheduler determines that no other thread needs to run on that CPU instead. Thread quanta are generally defined in terms of some number of clock ticks. If it doesn't otherwise cease to be runnable, the scheduler decides whether to preempt the currently running thread every clock tick. In order to observe and analyze this behavior, we have chosen the temporal resolution to be 1 us; a fraction of 1 scheduler clock tick. In Section 6.2.3, we describe the disadvantages of managing time as a fixed-step increasing variable and describe our solutions that significantly improve the generated state space and the efficiency of the analysis.

5.2.2   Modeling Temporal Partitioning

The place *Clocks* in Figure 6 holds the state of the node-specific global clocks. The temporal partition schedule modeled by these clocks enforces a constraint: component operations can be scheduled and component threads can be run only when their parent partition is active. Each clock token *NC* is modeled as a 3-tuple:

$$NC = <Node, Value, TPS_{Node}>$$ (1)

where, $Node_{NC}$ is the name of the computing node, $Value_{NC}$ is an integer representing the value of the global clock and $TPS_{Node_{NC}}$ is the temporal partition schedule on $Node_{NC}$. Each *TPS* is an ordered list of temporal partitions.

$$TP = <Name, Prd, Dur, Off, Exec>$$ (2)

Each partition *TP* (Eq. 2) is modeled as a record color-set consisting of a name $Name_{TP}$, a period Prd_{TP}, a duration Dur_{TP}, an offset Off_{TP} and the state variable $Exec_{TP}$.

Aggregate of such partitions can fully describe a partition schedule. Complete partition schedules are maintained per computing node. Figure 8 shows a C++-style struct representation of the temporal partition schedule. Each clock structure is modeled similarly in Stardard ML [62] as a color and these colored tokens are placed in the *Clocks* place (Figure 6).

```
struct Partition {
    string name;                      /* Name of the Partition */
    unsigned long duration;           /* Duration of the Partition */
    unsigned long period;             /* Period of the Schedule */
    unsigned long offset;             /* Offset of the Partition */
};

struct Clock {
    string name;                      /* Name of the Embedded Node */
    int value; ;                      /* Current value of the Clock */
    int next_tick;                    /* Next clock tick */
    vector<Partition> schedule;       /* Temporal Partition Schedule*/
};
```

Figure 8: Temporal Partition Schedule Data Structure

5.2.3 Modeling Component Thread Behavior

Figure 9 presents a snippet of our CPN, modeling the thread execution cycle. The place *Components* holds tokens that keep track of all the ready threads in each computing node. Each component thread *CT* is a record characterized by:

$$CT = < ID_{CT}, \; Prio_{CT}, \; O_{CT} > \tag{3}$$

where ID_{CT} constitutes the concatenation of strings required to identify a component thread in CPN, i.e., component name, node name and partition. Every thread is characterized by a priority ($Prio_{CT}$) which is used by the OS scheduler to schedule the thread. Component threads are maintained in a priority queue in the CPN model. When scheduling

33

a thread, a *Schedule_Thread* transition evaluates this priority queue and all the component message queues to find the highest priority component thread that is currently executing an operation or is ready to execute a new operation. If multiple component threads of equal (and highest) priority are ready to execute, then one of these threads is chosen at random and scheduled. When this thread consumes its quantum of CPU, the thread is enqueued to the back of the priority queue of threads and is not chosen again until all other competing threads are allowed a slice of CPU time.
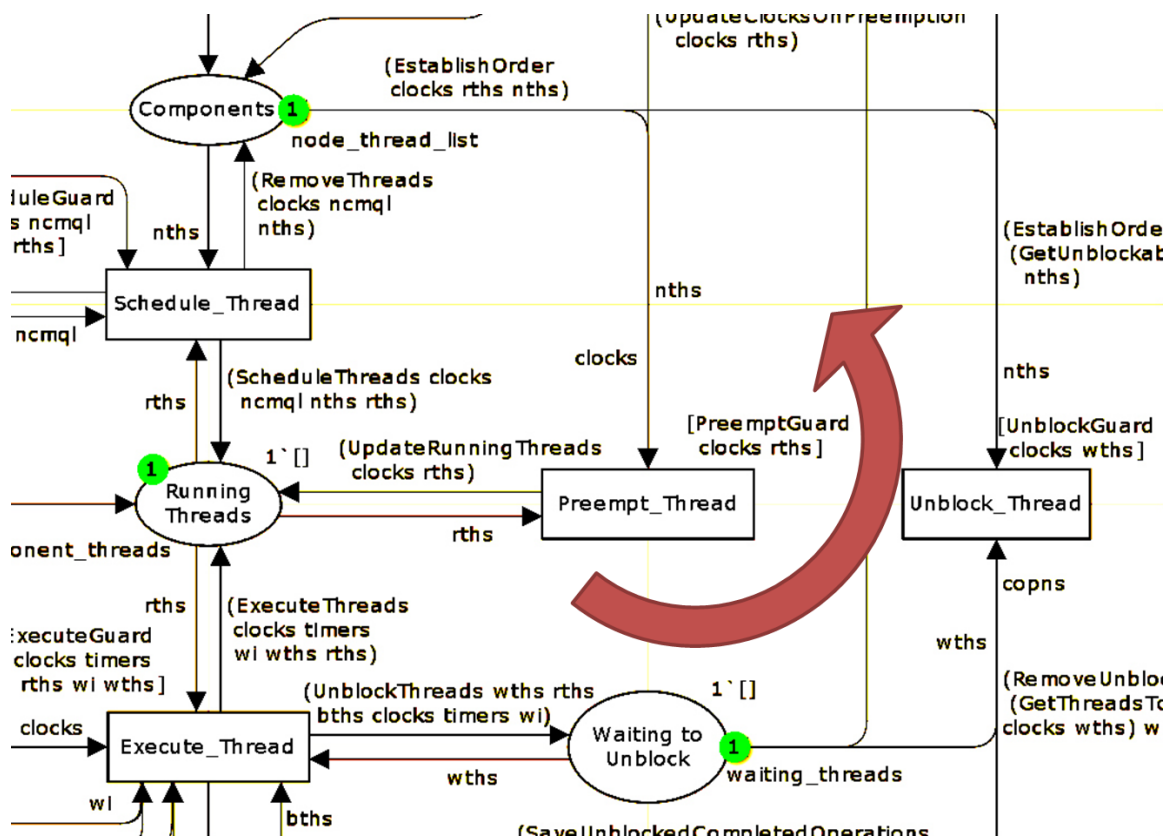
Figure 9: Component Thread Execution Cycle

If the highest priority (OS schedulable) thread is not already servicing an operation request, the next ready operation from the message queue is dequeued and scheduled for execution (represented by O_{CT}). Depending on the component scheduler, this operation

may be the highest priority, or may have the earliest deadline or may be the oldest request. The scheduled thread token is placed in *Running Threads*.

When a thread token is marked as running, the model checks to see if the thread execution has any effect on itself or on other threads. These state changes are updated using the transition *Execute_Thread* which also handles time progression. Keeping track of $Value_{NC}$, the thread is preempted at each clock tick. This transition loop, i.e., *Schedule_Thread -> Execute_Thread -> Preempt/Unblock_Thread -> Schedule_Thread ...* cycle repeats forever, as long as there are no system-wide deadlocks and some upper limit on the clock values isn't reached.

5.2.4 Modeling Component Operations

Every operation request *OP* made on a component C_x is modeled as a Standard ML *record* of the 4-tuple:

$$OP(C_x) = <ID_O, Prio_O, Dl_O, Steps_O>$$ (4)

where, ID_O is a unique concatenation of strings that help identify and locate this operation in the system (consisting of the name of the operation, the component, the computing node, and the temporal partition). Assuming a PFIFO operation scheduling scheme, the operation's priority ($Prio_O$) is used by the analysis engine to enqueue this operation request on the message queue of C_x. The completion of this enqueue implies that this operation has essentially been *scheduled* for execution. Figure 10 shows the operation scheduling cycle. The message queue tokens are placed in *CMQ* (i.e. Component Message Queues). The tokens in this place are a constraint on the *Schedule_Thread* transition and are used to calculate the subset of threads that are *runnable*. When threads execute, operation requests may be enqueued in component message queues as shown by this cycle. The transition *Enqueue_Operation* enqueues any operation requests that are sent by the currently executing

threads. These requests are enqueued onto the appropriate message queue in the chosen
scheduling scheme.
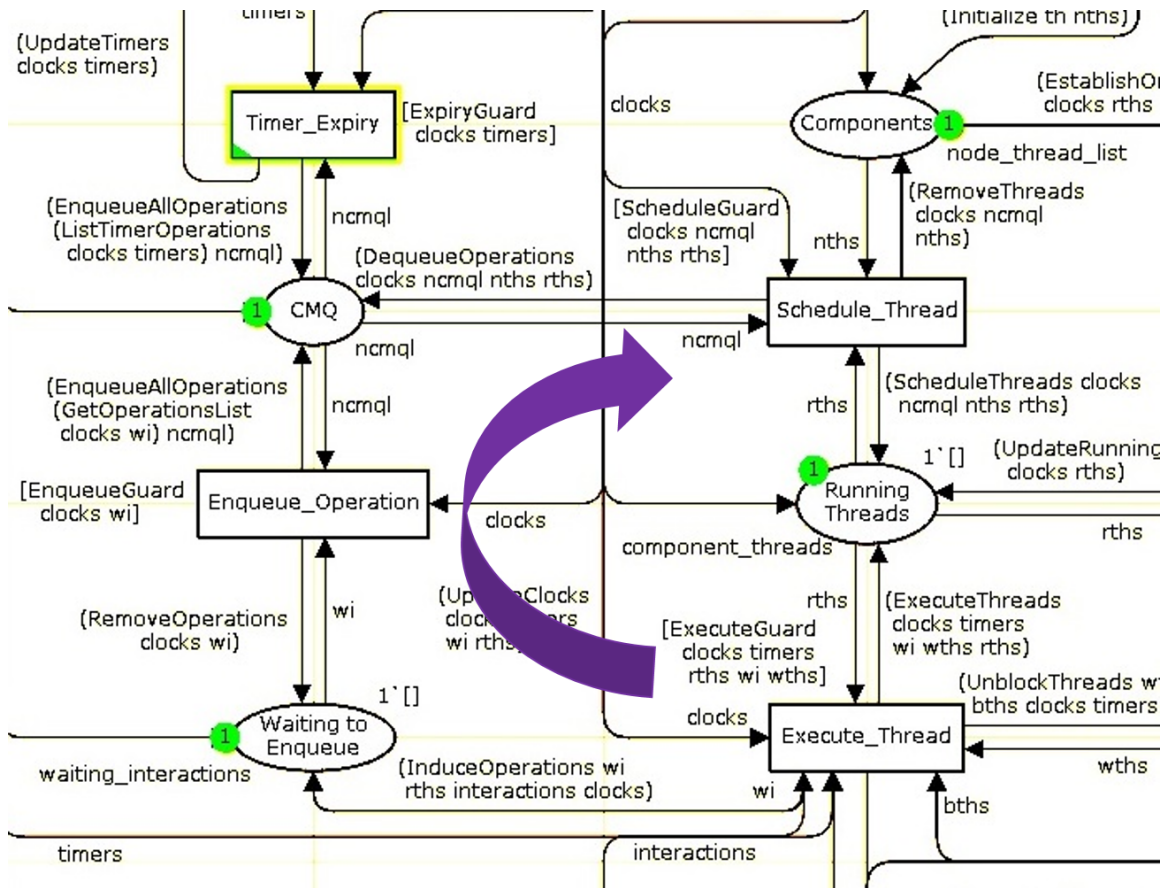
Figure 10: Component Operation Scheduling Cycle

Once enqueued, if this operation does not execute and complete before its fixed deadline
(Dl_O), its real-time requirements are violated. The completion of each operation request is
saved in the place *Completed_Operations*. This place is an *observer* that is simply notified
by the *Execute_Thread* transition every time the thread execution also completes the exe-
cuting operation. The tokens in Completed_Operations saves a snapshot of the operation's
properties including its operation's enqueue timestamp, dequeue timestamp, completion
timestamp and deadline. By analyzing the tokens in this place, deadline violations in an
interval of time can be detected.

Once an operation request is dequeued, the execution of the operation is modeled as a transition system that runs through a sequence of *steps* dictating its behavior. Any of these underlying steps can have a state-changing effect on the thread executing this operation e.g. interactions with I/O devices on the component-level could block the executing thread (for a non-deterministic amount of time) on the OS-level. Therefore, every component operation has a unique list of steps ($Steps_O$) that represent the sequence of local or remote interactions undertaken by the operation. Each of the *m* steps in $Steps_O$ is a 4-tuple:

$$s_i = < Port, \ Unblk_{s_i}, \ Dur_t, \ Exec_t > \tag{5}$$

where $1 \leq i \leq m$. *Port* is a *record* representing the exact communication port used by the operation during s_i. $Unblk_{s_i}$ is a list of component threads that are unblocked when s_i completes. This list is used, e.g., when the completion of a synchronous remote method invocation on the server side is expected to unblock the client thread that made the invocation. Finally, temporal behavior of s_i is captured using the last two integer fields: Dur_t is the worst-case estimate of the time taken for s_i to complete and $Exec_t$ is the relative time of the execution of s_i, with $0 \leq Exec_t \leq Dur_t$.

Consider the simple RMI application show in Figure 11. The application has two components, a client and a server. The client component is associated with a periodic timer that triggers a sequence of interactions between the two components. When the client timer expires, a timer operation is enqueued on the client's operation queue. When scheduled, the client executor thread executes this operation, which makes an RMI call to the server component. Once the query is made, the client thread is effectively blocked till a response is received. The server thread that produces this response may not be scheduled immediately due to the constraints of temporal partition scheduling and other thread scheduling delays. Once the RMI operation is completed on the server, the client thread is unblocked.

In the above example, the duration of time for which the client is blocked, is dependent
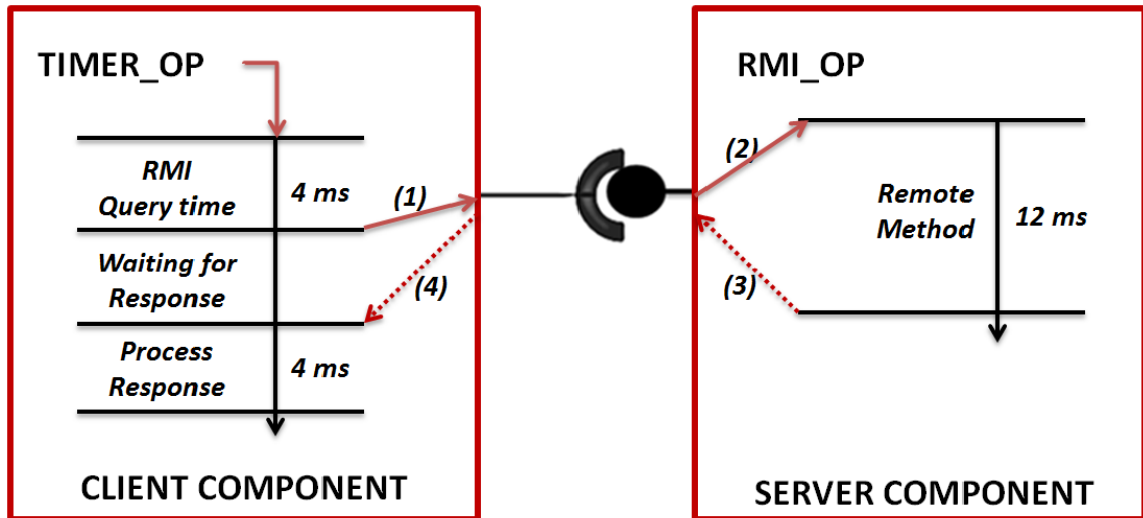
Figure 11: RMI Application

on, among several factors, what happens inside the remote method on the server. This remote method could either simply take up CPU time, interact with the underlying framework or interact with other components in the application. To capture such interaction patterns, the *step* color-set is defined in CPN. In this example, two *operation* tokens are required to describe the operations handled by the components: a client side timer operation and a server side RMI operation. A sample client timer operation is shown in Figure 12.

1`[{node="**Beaglebone_111**", period=100000, offset=0,
        operation={node="Beaglebone_111", component="**Client_Component**", operation="**TIMER_OP**",
                priority=50, deadline=80000, enqueue_time=0,
                    steps=[{kind="LOCAL", port="LOCAL", unblk=[], exec_time=0, duration=**4000**},
                            {kind="**CLIENT_RMI**", port="client_port", unblk=[], exec_time=0, duration=0},
                            {kind="LOCAL", port="LOCAL", unblk=[], exec_time=0, duration=**4000**}]}}]

Figure 12: RMI Application - Client Timer Operation

This timer operation runs on the client component with a priority of 50, and a deadline of 80 ms. The business logic of this operation consists of a single RMI call that takes 4 ms to send out the query after which it blocks the executing client thread. After the client thread runs for time $t = q\_t$, the client thread is moved to a blocked state and an RMI operation is

38

induced on the server side. The client side thread remains blocked until the server thread completes executing the remote method. Once the server thread completes execution, it sends the response of the RMI back to the client. The model takes note of how long the client has been blocked by using the time stamp at which it received a response. The client thread runs for an additional 4 ms to process this response before it marks completion. The token for the server RMI operation is shown in Figure 13. Note that all time measurements in this token are in micro-seconds i.e. a step duration of 4000 implies 4 ms of activity. The requested RMI operation is run on the server component with a priority of 50 and a deadline of 15 ms. The deadline of this operation cannot be worse than the deadline of the client side operation that initiated the interaction. If this operation delays past 80 ms, a client side deadline violation is detected as the client thread is blocking for longer than expected.

```
1`{node="Beaglebone_112", component="Server_Component", operation="RMI_OP",
    priority=50, deadline=15000, enqueue_time=0,
              steps=[{kind="LOCAL", port="LOCAL",
              unblk=[{node="Beaglebone_111", component="Client_Component', port="client_port"}],
              exec_time=0, duration=12000}]}}
```

Figure 13: RMI Application - Server Operation

5.2.5 Modeling Component Interactions

In our earlier RMI example, the client is periodically triggered by a timer to make a remote method call to the server. When the client executes an instance of this operation, a related operation request is enqueued on the server's message queue. In reality, this is handled by the underlying middleware. Since the details of this framework are not modeled, the server-side request is captured as an *induced operation* that manifests as a consequence of the client-side activity. Tokens that represent such design-specific interactions are maintained in the place *Component Interactions* (Figures 6,14) and modeled as

shown in equation 6. The interaction *Int* observed when a component C_x queries another component C_y is modeled as the 3-tuple:

$$Int(C_x, C_y) = < Node_{C_x}, Port_{C_x}, O(C_y) > \qquad (6)$$

When an operational *step* in component C_x uses port $Port_{C_x}$ to invoke an operation on component C_y, the request O_{C_y} is enqueued on the message queue of C_y.
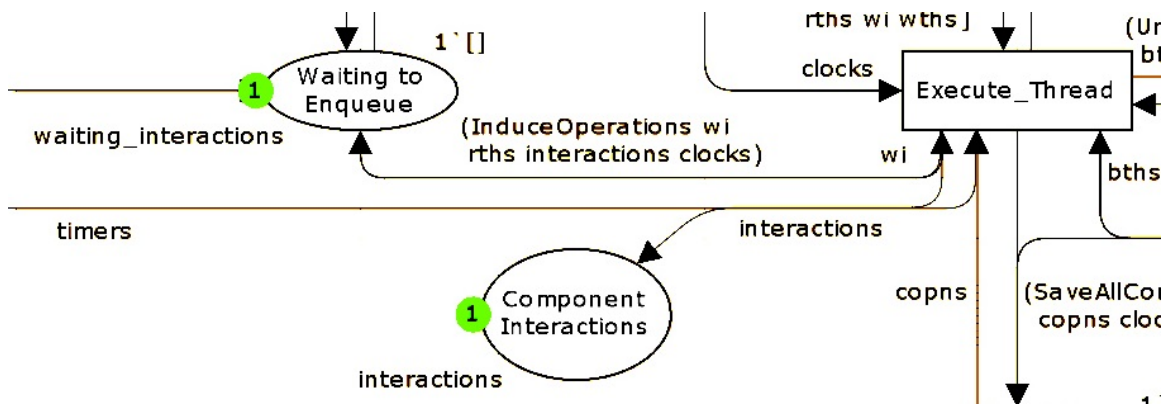
Figure 14: Operation Induction

Every *interaction* token contains an interaction port and an operation. The transition *Execute_Thread* observes the activity on the currently running thread. When executing the model, if a particular step executed by a component thread would, on completion, request the services of another component, a token is placed on the *Waiting to Enqueue* place. So, once the client thread pushes out an RMI query, an operation needs to be induced on the server queue. So an *interaction* token for this communication is constructed. The model waits for the RMI call on the client side to complete, at which point it places the operation *i_op* on the server message queue. This induction is represented in Figure 15.

```
1`{node="Beaglebone_111", port="client_port",
 operation={node="Beaglebone_112", component="Server_Component", operation="RMI_OP",
            priority=50, deadline=15000, enqueue_time=0,
            steps=[{kind="LOCAL", port="LOCAL",
            unblk=[{node="Beaglebone_111", component="Client_Component', port="client_port"}],
            exec_time=0, duration=12000}]}}}
```

Figure 15: Operation Induction Token

5.2.6 Modeling Timers

DREMS components are inactive initially; once deployed, a component executor thread is not eligible to run until there is a related operation request in the component's message queue. To start a sequence of component interactions, periodic or sporadic timers can be used to trigger a component operation. In CPN, each timer TMR is held in the place *Timers* and represented as shown in Eq. 7. Timers are characterized by a period (Prd_{TMR}) and an offset (Off_{TMR}). Every timer triggers a component using the operation request O_{TMR}.

$$TMR = <Prd_{TMR}, Off_{TMR}, O_{TMR}> \tag{7}$$

When the component's timer expires, a timer callback operation is placed on the component message queue. When the component executor thread is picked by the OS scheduler, this operation is dequeued and the timer callback is executed. In CPN, timer operations are modeled as shown in Figure 16.

All component timers are expressed as separate tokens and initialized in the *Timers* place. It is important to note that the enqueue operation does not happen until the appropriate partition is active. This is because the component-specific thread responsible for enqueueing (or dequeuing) incoming operations is also affected by temporal partitioning.

41
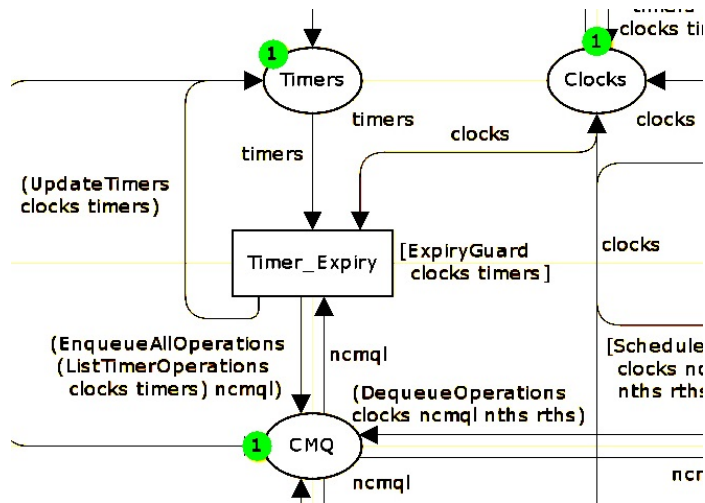
Figure 16: Timer Operations

5.3   Modeling Component Operation Business Logic

5.3.1   Problem Statement

Consider a set of component-based applications deployed on distributed hardware. Each application consists of groups of components that interact with each other and also with the external environment, e.g., I/O devices, other applications, underlying middleware etc. Each component exposes a set of interfaces through which external entities can request *operations*. As mentioned earlier, an operation is an abstraction for the different tasks undertaken by a component. These operations are exposed through ports and can be requested by other components. When an operation is requested, the request is placed in the component's message queue and eventually serviced. When ready, the business logic of the operation, i.e., a local callback is executed. This piece of code represents the brains of the operation. The goal here is to be able to model this business logic, for every component operation, effectively as part of the design model, including temporal estimates such as worst-case execution times for individual code blocks, so that the model can be translated into appropriate data structures in our CPN analysis model.

5.3.2 Challenges

The execution of component operations service the various periodic or aperiodic inter-
action requests coming from either a timer or other connected (possibly distributed) com-
ponents. Each operation is written by an application developer as a sequence of execution
steps. Each step could execute a unique set of activities, e.g., perform a local calculation
or a library call, initiate an interaction with another component, process a response from
external entities, and it can have data-dependent, possibly looping control flow, etc. The
behavior derived by the combination of these steps contribute to the worst-case execution
of the component operation. The behavior may include non-deterministic delays due to
component interactions while being constrained by the temporally partitioned scheduling
scheme and hardware resources. The challenge here is to identify a language that would
enable the description of potentially dynamic behavior realized in a component operation.
The modeling aspects emerging from this challenge will have to propagate to any timing
analysis model that studies the system. This is true because any non-deterministic delays
such as blocking times need to be accounted for when analyzing the temporal behavior.

5.3.3 Outline of Solution

The business-logic model of a component operation requires to be completely inte-
grated into our CPN modeling methodology. This means that the model, however complex,
needs to be translated into some token data structure in CPN. This is our primary constraint.
The CPN analysis model needs to know how an operation is structured, i.e., what are the
sequential steps in the code, along with WCET on each step. Lastly, since the CPN model
does not model or simulate component data management, data-dependent conditional state-
ments in the business-logic model were avoided or abstracted away. Following these rules,
we designed a language for describing the component operation business logic. Each com-
ponent operation model is then attached to a component port or timer in the main design
model and enriches the model with refined details about the workings of the operation.

In summary, this model is capable of representing several types of code blocks including local function calls, remote procedure calls, outgoing port-to-port interactions, incoming port-response processing, and bounded loops.

The execution of component operations service the various periodic or aperiodic interaction requests coming from either a timer or other connected (possibly distributed) components. Each operation is written by an application developer as a sequence of execution *steps*. Each step could execute a unique set of activities, e.g., perform a local calculation or a library call, initiate an interaction with another component, process a response from external entities, and it can have data-dependent, possibly looping control flow, etc. The behavior derived by the combination of these steps contribute to the worst-case execution of the component operation. The behavior may include non-deterministic delays due to component interactions while being constrained by the temporally partitioned scheduling scheme and hardware resources. This section briefly describes the various aspects of this behavior specification that are general enough to be applicable to a range of component-based systems.

Figure 17 shows the Extended Backus-Naur form representation of the grammar [49] used for modeling the business logic of component operations. The symbol *ID* represents identifiers, a unique grouping of alphanumeric characters, and the symbol *INT* represents positive integers. Each operation is characterized by a unique name, a priority, and a deadline. The priority is an integer used to resolve scheduling conflicts between operations *provided* by the same component when multiple messages from other entities are received. This priority is different from the executor thread's OS-level priority; operation's priorities are used by the component-level scheduler to find the next operation to be executed by a component executor thread. The deadline of the operation is the worst-case time that can elapse after the operation is marked as *ready* and the completion of the operation. The business logic of every component operation is modeled as a sequence of steps, each with

44

```
business_logic          =   'do', ws, operation_name, ws
                            '{', { functional_step }, '};' ;
operation_name          =   ID ;
functional_step         =   {sequential_code_block | rmi_call | ami_call | publish| loop} ;
sequential_code_block   =   'LOCAL' INT, ';' ;
rmi call                =   'RMI', ws, receptacle_port, '.', remote_operation, ('[' query_time, processing_time '];')?;
ami call                =   'AMI', ws, receptacle_port, '.', remote_operation, ('[' query_time, processing_time '];')?;
publish                 =   'PUBLISH', ws, publisher_port, '.', topic, ('[', publish_time, '];')? ;
loop                    =   'LOOP', ws, '[', count, ']', ws,
                            '{', {functional_step}, '};' ;
receptacle_port         =   ID ;
remote_operation        =   ID ;
publisher_port          =   ID ;
topic                   =   ID;
query_time              =   INT ;
processing_time         =   INT ;
publish_time            =   INT ;
count                   =   INT;
ws                      =   ? white space characters ?
identifier          =       alphabetic character, { alphabetic character | digit } ;
Integer             =       digit, {digit} ;
digit               =       "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Figure 17: Grammer for the Business Logic of Component Operations

known worst-case execution time. We broadly classify these steps into (1) blocks of sequential code, (2) peer-to-peer synchronous and asynchronous remote calls, (3) anonymous publish calls, (4) blocking and non-blocking I/O interactions and (5) bounded loops.

Business Logic models represent the sequence of steps that are executed in a component operation. These component operations are executed when the component is triggered which can happen in three ways: (1) the expiry of a timer executes a callback, (2) the reception of message on the subscriber port, and (3) the reception of a method request from a remote client on a server port. When one of these events occurs, the corresponding operation is enqueued on the message queue and eventually handled. When the operation is executed, the component can use any and all outgoing ports at its disposal to publish messages or query other remote servers, as shown in Figure 17. Additionally, the operation can also execute local non-blocking function calls that perform any required computations. For each such step, the grammar enables the integration of timing properties. With RMI calls, *query_time* is an optional annotation that represents the worst-case estimated duration

45

of time taken by a client port to send out a request to the serving component. This time can be used to include network buffering delays or any other pre-processing steps enforced by the infrastructure before the request actually leaves the client component. Similarly, *processing_time* represents the duration of time after the client component receives the response from the server when any post-processing steps are executed by the infrastructure. These delays are optional and can be encapsulated within *LOCAL* blocks on either side of the call. If these expected delays are set to zero, the analysis will execute these interactions in a single synchronous step taking no time. However, in reality these steps still take a non-zero amount of time to execute. Therefore, if such metrics are not known then these values can be set to zero and an overall worst-case execution time can be set per operation. This is the maximum amount of time that can elapse after the component operation has begun to execute. This time will include all component interactions and network delays that affect the operation's execution.

Figure 18 shows a sample business logic model conforming to this grammar. The Data Acquisition Module is a periodically triggered I/O component, i.e., this component receives a stream of sensor information from various sensor devices e.g. inertial measurement units (IMU) and GPS modules. This component packages this information and publishes sensor state as a message to all subscribers, e.g., components that implement controllers. Figure 18 shows the translation from the conceptual understanding of the workings of this component operation to the abstract business logic model that is then translated into CPN tokens (Figure 19), as described in Section 5.2.4.

Every time the above timer expires, this *sensor_read_timer* operation is enqueued onto the component message queue (Figure 10). When the *Data_Acquisition_Module* component is scheduled, this operation is dequeued from the message queue and is marked for execution. The *Execute_Thread* transition fires, this component thread executes i.e. each operation step in the component operation proceeds to sequentially consume CPU. So, first the LOCAL code block executes for 15 milliseconds. Every 4 milliseconds (the default
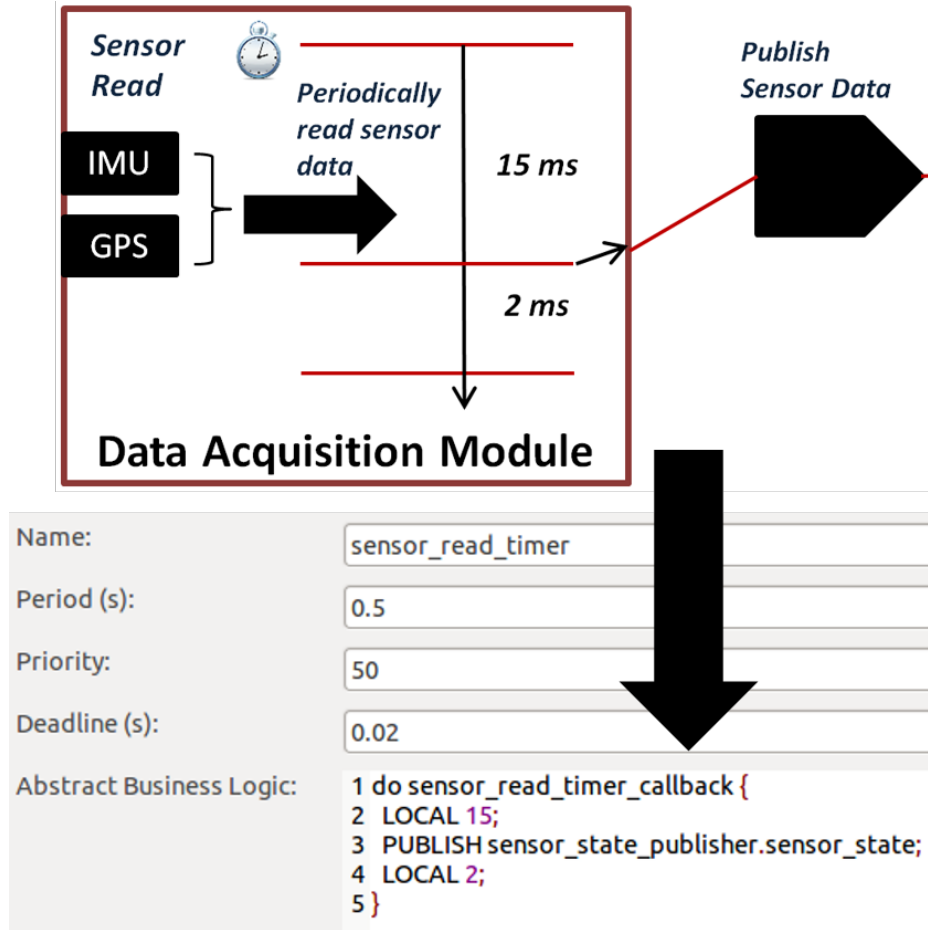
Figure 18: Sample Business Logic Model

The figure contains a "Data Acquisition Module" with:
- Sensor Read (IMU, GPS)
- Periodically read sensor data, 15 ms
- 2 ms
- Publish Sensor Data

And a properties panel:

Name:	sensor_read_timer
Period (s):	0.5
Priority:	50
Deadline (s):	0.02
Abstract Business Logic:	

```
1 do sensor_read_timer_callback {
2   LOCAL 15;
3   PUBLISH sensor_state_publisher.sensor_state;
4   LOCAL 2;
5 }
```

clock tick), the scheduler checks to see if this thread needs to be preempted in favor of any other higher priority ready thread. If no other higher priority threads are ready to execute, this thread continues to consume CPU and the *exec_time* attribute of this step is incremented. When *exec_time = duration*, this step is removed from the operation and the next step begins execution. Here, the component uses its publisher port to send out a sensor_state message. Since no other delays are detailed for this step, the analysis model executes this step in zero time. When this step completes, the *Execute_Thread* transition enqueues an operation on all subscriber components' message queues following the

```
1`[{node="Beaglebone_111", period=500000, offset=0,
      operation={node="Beaglebone_111", component="Data_Acquisition_Module", operation="sensor_read_timer",
                 priority=50, deadline=20000, enqueue_time=0,
                    steps=[{kind="LOCAL", port="LOCAL", unblk=[], exec_time=0, duration=15000},
                                {kind="PUBLISH", port="sensor_state_publisher", unblk=[], exec_time=0, duration=0},
                            {kind="LOCAL", port="LOCAL", unblk=[], exec_time=0, duration=2000}]}}]
```

Figure 19: CPN Business Logic Representation

interaction rules described in Section 5.2.5. Lastly, a LOCAL code block consumes 2 mil-
liseconds of CPU and the operation is marked as complete. The clock value on this node
when the operation completes is saved as a completion_time of this operation.

5.4   Modeling Component-based Cyber-Physical Systems

A large subset of modern DRE systems are also Cyber-Physical Systems (CPS) e.g.
flight controllers [81], traffic light controllers [39] etc. In these systems, the runtime ap-
plication can consist of a number of sub-systems including a list of sensors, actuators, and
controllers that interact with each other and govern the dynamics of the *plant*, i.e., the phys-
ical state of the system. For instance, a DREMS-style sensor component in a flight control
scenario may interact with low-level hardware, e.g., GPS, inertial measurement units, cam-
eras, etc., receive a stream of information, sample sensor state from this stream and publish
this information at a defined period. A high-level controller may receive this sensor data
periodically and perform some PID control, commanding a remote actuator component
with new settings, e.g., the deflection of control surfaces, that in turn cause changes in the
attitude of an aircraft, elevations in pitch etc. The actuator periodically receives these com-
mands and appropriately modifies the state of the low-level hardware, e.g., driving a set
of motors for a defined duration of time so that an aircraft control surface causes airplane
take-off.

When modeling CPS sensors, the analysis considers two cases: request-response style

queries and periodic sensor streams. When using request-response queries, a sensor software component operation is triggered, e.g., through a component timers, and this operation actively queries the sensor hardware for updated state. The query can be blocking or non-blocking, and the sensor software component publishes this updated state to other interested components. When using periodic sensor streams, the component is triggered by the arrival of sensor data (that is streaming) and the component publishes the received data sample.

Similarly, an actuator component may receive commands to actuate, e.g., deflecting the flaps on the trailing edge of an aircraft wing. This is the second *I/O component* that directly interacts with hardware and influences the state of the system. For the sake of modeling simplicity, actuation commands are modeled as non-blocking write operations performed by software components to control low-level hardware. By modeling these interactions as separate from other local code blocks or function calls, the business logic model is enriched with a broader set of actions that can be used to identify the source of timing delays in a component operation. This section describes the integration of these designs into both the business logic model and the CPN analysis.

Figure 20 shows the integration of CPS sensor models into the CPN analysis model. The sensor hardware is modeled in the *Sensor* place; each sensor token is associated with a period representing the periodicity of the state updates. Each sensor state is updated by the *Sensor_Update* transition periodically and a new *Sensor_State* is recorded. This sensor state is either queried by a component operation or received as part of a sensor stream. In case of blocking read operations, the querying operation blocks until a new sensor state is available in the Sensor_State place.
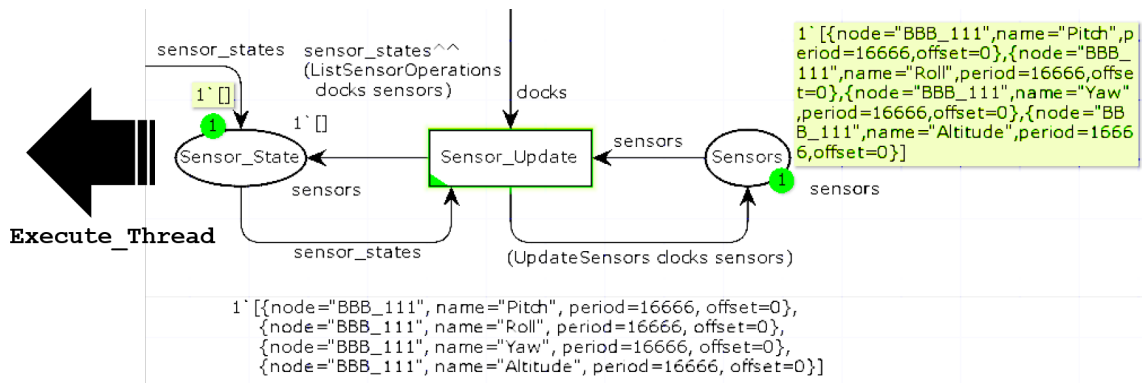
Figure 20: CPN Analysis Model - CPS Sensors

The business logic of the component operations can read the state of the variables in *Sensor_State* as required when performing computations. Since the CPN analysis does not model data or data flow, only the name of the required sensors is captured in the business logic. Thus, the business logic model, as described in Section 5.3 is revised as shown in Figure 21.

Figure 21: Modeling CPS - Business Logic Integration

The business logic model integrates two types of sensor read operations and one actuator write operation. The sensor read operation can be blocking or asynchronous. In blocking reads, the operation does not continue execution if the required sensor state is

50

unavailable in *Sensor_State* place. When a new state is available, the currently blocked operation consumes the sensor token and proceeds execution i.e. the worst-case time taken to perform the sensor read is dependent on the periodicity of the sensor update. This is similar to a request-response style query, i.e., a sensor component requests updated state information from a sensor device, and the sensor responds as soon as it is updated. In asynchronous read operations, if an updated sensor state is unavailable, the operation uses the latest state, i.e., the previous state of the sensor obtained from a previous read operation. Asynchronous sensor read operations do not take any time and the CPN analysis does not assume any faulty behavior on the part of the sensors, i.e., the sensor state is strictly updated periodically without any delays. As for actuators, the most common means to control an actuator, e.g., servo motor, is to write a value to the corresponding actuator system variable(s), e.g., toggling the state of a set of general purpose input-output (GPIO) hardware pin(s). Such system variables or hardware pins are assumed to be always ready to accept new write operations.

The above models are quite simplistic. There is no way for the analysis model to know if the received state information is stale or not. The component operation, at runtime, may decide not to publish sensor state if the received state information is stale. Such a choice cannot be modeled with this business logic since there is neither model of data (values of local variables or sensor state) nor any model of conditional statements. Thus, in such cases, the results of the analysis can lead to gross overestimates as the analysis will only consider the case when the publish always happens, i.e., the worst-case scenario.

CHAPTER VI

STATE SPACE ANALYSIS AND VERIFICATION

The state of a dynamic system is represented by a minimal set of variables, called state variables, that fully describe the system and its response to any given set of inputs. This minimum set of variables, $s_i(t), i = 0, 1, 2, ..n$ along with knowledge of those variables at an initial time t_0 and the system inputs for time $t > t_0$, are sufficient, in a deterministic system, to predict the future system state and outputs for all time $t > t_0$. This asserts that the dynamic behavior of a state is completely characterized by the set of state variables $s_i(t)$.

CPN Tools uses a built-in *state space* analysis tool to generate a bounded state space from an initialized CPN model. Here, the state space is a directed graph structure where the vertices, called states, represent a unique system state. An edge between two states represents the transition from one state to another. If a state S can non-deterministically transition into K possible future states, then S becomes the root of a K-ary tree. The state of the system in our case is a record of all the places in the CPN i.e. the token values in every place of the CPN model. This record is therefore a snapshot of the token configuration of the net and represents its execution state. State space generation is a process of generating this directed graph, from some initial state. For pragmatic reasons, we generate a bounded state space, i.e., a graph structure bounded by some rule, e.g., $t_i < t_{bound}$, where t_i is some global variable representing time. It must be noted here that CPN Tools supports the notion of global variables; one such variable is used to establish this bound. In this case, the state space will contain only nodes where the state variable t_i is less than some upper bound t_{bound}. Alternately, the rule can be to generate a state space as long as the component message queue size is under 50 waiting requests.

To illustrate the state space analysis of a component-based application using our CPN,

we consider a simple example – consider three equal-priority components executed on a single device. Each component has a periodic timer that fires every 10 ms and triggers the execution of a block of code. Each component maps to an executor thread and these three threads are scheduled concurrently with all other threads in the system. In this example, there are no other component threads or system-level threads considered. Based on the OS scheduling scheme, these components, with equal priority, are scheduled using round-robin policy. Since all three timers have the same period and the relative offsets are zero, the three timers fire concurrently and the respective component threads are marked as 'ready' at the exact same time. Since all three component threads are ready to execute the same times, there are 3! possible thread execution orders in the worst case when following round-robin scheduling, assuming single core execution.

Figure 22 shows a bounded state space generated in CPN Tools for this component assembly. The component threads have the same priority and are executed in the same device. Each component is triggered with a 100 Hz periodic timer and all three timers are synchronized to illustrate non-determinism. The round robin scheduling quantum is set to 4 milliseconds. 'Mark' is a state space query function that provides the *marking* of a place in a particular state space node. There are 6 branches from the initial state of the system as the model realizes the 6 possible behaviors. Each node in this state space is annotated with a state space ID and also a pair of integers in the format *"p:c"*, where p refers to the number of parent nodes and c refers to the number of child nodes. This figure also shows results of a *state space query*. Specifically, the query finds the *marking* on the *Completed_Operations* place in two different state space nodes, 35 and 37. In node 37, Timer_3_operation is the first operation to complete as Component_3 is the first ready thread chosen by the OS. In node 35, Timer_1_operation is the first operation to complete as Component_1 is the first ready thread chosen by the OS. This illustrates the tree of possible behaviors that is represented by the state space starting from an initial state. The goal of state space analysis is to search this tree of possibilities to identify a single execution trace, i.e., a single branch

53

in this tree that either satisfies or negates a system property, e.g., that the deadline of one of these timer operations is violated.
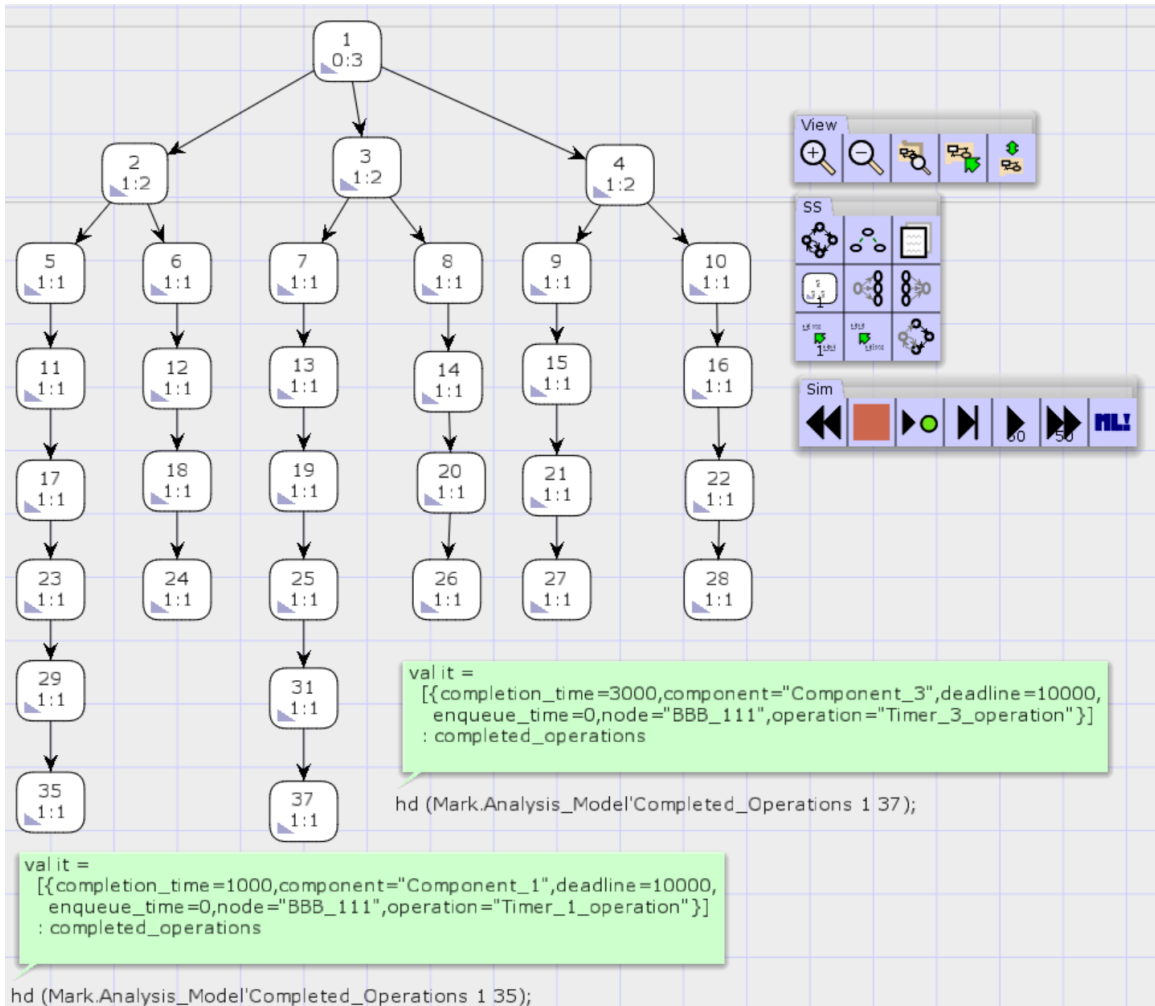

Figure 22: Bounded State Space for a Multi-component Timer example

6.1   Searching the State Space

CPNTools' inbuilt state space analysis tool comes with a programming interface – a set of function that can be used by a user to query a generated state space. One of the many

available functions is the *SearchNodes* function, as shown in Figure 23. This function tra-
verses the nodes of the state space and at each node, evaluates a predicate and accumulates
a list of nodes that satisfy this predicate.

```
SearchNodes(Area, Pred, Limit, Eval, Start, Comb)
  begin
    Result := Start; Found := 0
    for all n ∈ Area do
      if Pred(n) then
        begin
            Result := Comb(Eval(n),Result)
            Found := Found + 1
          if Found = Limit then
              stop for-loop
        end
      end
  end
```

Figure 23: SearchNodes function provided by CPNTools

There are six parameters provided to this search function. *Area* refers to the search area,
i.e., the part of the state space that needs to be searched. Often, the search area is the entire
graph but it is possible to provide a subset of the graph, e.g., a list of strongly connected
components [1]. The second argument, *Pred*, specifies a predicate function that evaluates
each node and produces a boolean result. All nodes that evaluate to false are ignored and
all nodes that evaluate to true are retained for further analysis. The third argument, *Limit*
is an integer referring to how many times a predicate should evaluate to true before the
search should terminate. If this limit is infinite, the entire state space is always searched.
Eval is an evaluation function that is executed on all state space nodes that satisfy the
predicate function, e.g., an evaluation function to find the execution time of an operation
when its predicate function detects a deadline violation. Lastly, *Start* is the initial value
of the result and *Comb* is a combination function that accumulates each new result from

---

[1] A directed graph G is strongly connected if every vertex is reachable from every other vertex in the graph.
A strongly connected component is a maximal strongly connected subgraph of G, i.e., no additional edges or
vertices from G can be included in the subgraph without breaking its property of being strongly connected

the evaluation function with prior results. In our analysis, the size of the state space can grow exponentially, e.g., as a function of time. The combination function is a constant time operation as it simply appends a new node to the end of the accumulated results list, given the searching algorithm a time complexity of $O(n)$ where n is the number of nodes in the state space. However, if the combination function is dependent on the size of the input, then it directly affects this search complexity.

The rest of this chapter details how a bounded state space can be used to analyze DREMS applications for deadline violations, response times predictions etc.

6.1.1 Deadline Violations

A *deadline violation* refers to a system state where the execution time of a component operation has exceeded its deadline. The operation may violate its deadline without beginning execution since wait times in the message queue count towards the total delay from the arrival of the message to the completion of the corresponding operation. The SearchNodes function in CPNTools is quite generic and can be easily applied to our analysis model to identify such violations in the state space. For all operations, either completed or waiting for execution, it is sufficient to execute the predicate *current_time − operation.enqueue_time > operation.deadline*. All nodes that satisfy this predicate are nodes that represent deadline violation states.

Alternatively, by adding *observer places* to our timing analysis model, i.e., places that passively observe the system and accumulate tokens when certain conditional transitions execute, deadline violations can be recorded as the model is executing. A *Deadline_Violation* (Figure 24) transition fires at any point in time when the guard *dl_guard* is satisfied and arc bindings exist with its input places. The transition observes the states of the currently running threads and the component message queues to identify deadline violations on operations that are either executing or waiting to execute. The *dl_violation* tokens are collected in a place called *Late Operations* (*LO*),

$$LO = <Node_{name}, O_{name}, O_{ST}, O_{DLT}> \qquad (8)$$

where operation O_{name} executing on computing node $Node_{name}$ started at time O_{ST} and violated its deadline at time O_{DLT}. Since the component-level scheduler uses a non-preemptive scheme, this operation is still run to completion after the violated deadline. Delays like these propagate to the waiting operations in the message queue.
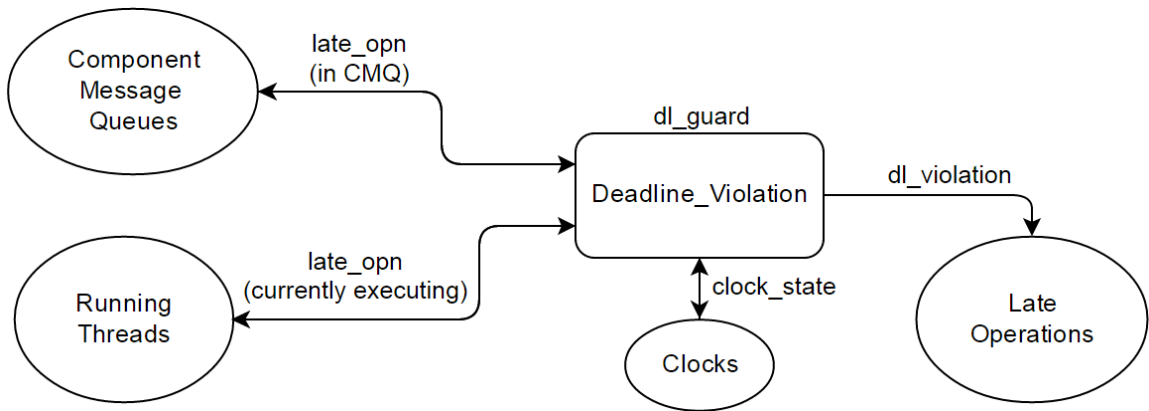
Figure 24: Deadline Violation Observer place

6.1.2 System-wide Deadlocks

System-wide deadlocks are caused by bad application design, e.g., a set of executing threads are indefinitely blocked on each other; a server operation invoking an RMI on an already blocked client component. Such scenarios are typically due to cyclic dependencies that are undetected during system design. Using state space analysis, such deadlocks can be detected in several ways.

One of the ways is to check if all of the component threads are blocked in any state space node, i.e., the number of tokens in the place *Blocked_Threads* is equal to the total number of component threads. In CPNTools, this is expressed using the Standard ML query as shown in Figure 25. This query consists of a function definition and a function call. First,

the function *AllThreadsBlocked* is defined, that returns true if the length of the marking in the place Blocked_Threads in the Analysis_Model page at the nth state space node, is equal to total number of component threads in the system. This function is used as the predicate function in the subsequent call to the SearchNodes function. The SearchNodes function is invoked with a set of arguments that request the search of the entire generated state space, to find the subset of state space nodes where the AllThreadsBlocked predicate is true, and to accumulate these nodes and save the result in the SystemWideDeadlock variable. This variable, if non-empty, indicates the presence of a system-wide deadlock. Using the nodes in the list, a trace can be generated to find the sequence of operations that lead to the deadlock. Using this trace, any implicit circular dependencies can be detected.

```
(* Predicate function that checks to see if the number of tokens in the place
   Blocked_Threads is equal to the total number of component threads in *)
fun AllThreadsBlocked n =
    (length (Mark.Analysis_Model`Blocked_Threads 1 n)) = NUM_COMPONENT_THREADS;

(* Search the state space and identify all state
   space nodes where a system-wide deadlock has occurred *)
val SystemWideDeadlock =
        SearchNodes (
            EntireGraph,
            fn n => (AllThreadsBlocked n),
            NoLimit,
            fn n => n,
            [],
            op ::);
```

Figure 25: SML Query to detect system-wide deadlocks caused by (blocking) cyclic dependencies between components

6.1.3   Response-time Analysis

Response time analysis identifies the worst-case time taken for the system to generate a desired output signal after an input trigger has been provided, e.g., time taken for the emergency braking system on an automated train controller to respond to sensory input.

A component-based system can have a variety of triggers but in this context, a trigger is considered as any operation request received by a component. The response to this trigger is the completion of some other operation at a future time instant, e.g., activating the brakes, after the occurrence of the trigger. With state space analysis, it is possible to identify the worst-case response times for a $(trigger\_operation, response\_operation)$ pair by first obtaining response times for all trigger-to-response paths and finding the maximum.

Similar to deadline violation detection, using the SearchNodes function, this can be accomplished as follows. The predicate function for the search tests all state space nodes for the completion of the response_operation. The evaluation function scans the list of completed operations in all state space nodes where a response operation was the last operation marked as completed. In this list, by identifying the trigger operation and response operation, the response time is calculated as the difference $Response\_Operation_{cmpl\_time} - Trigger\_Operation_{enq\_time}$. These results are accumulated by the combination function and the maximum response time is calculated from the resultant list. The time complexity of this search is again linear with respect to the size of the state space but the size of the state space itself can be exponential. The size of the accumulated results list is at most half the size of the state space since the trigger and the response are always in separate state space nodes. Finding the maximum trigger-to-response time within the resultant list is also done in linear time.

This result, as with other state space analysis results here, assumes ideal functional behavior for all operations, i.e., the completion of the operation always provides the desired results. Since the business logic model for operations does not encode data-dependent behavior and conditional execution, the completion of a response operation is no indication about the correctness of the response, merely its timeliness. It is therefore also not possible to differentiate between response "types" as all responses are seen as equivalent.

6.2   Modeling and Analysis Improvements

6.2.1   Problem Statement

The CPN analysis work presented in [51], i.e., the original approach developed and described in the previous section, has some limitations. The clock values in the distributed set of computing nodes progress by a fixed amount of time regardless of the pace of execution. This is one of the primary causes of state space explosion since many of the intermediate states between *interesting* events, though uneventful, are still recorded by the state space generation. For instance, in a temporal partition spanning 100 ms, even if a thread executes for 5 ms and the rest of the partition is empty, then if the clock progresses at a 1 ms rate, a 100 states are recorded in the state space when there are at most 5-7 interesting events in this interval. For a larger set of distributed interacting components, this can become a problem. Also, for distributed scenarios where multiple instance of a set of applications are executed in parallel, on independent computers, our CPN modeling methodology isn't efficient, leading to a tree of parallel executions even when the distributed computers are independent i.e. the computers can be synchronously progressed. The goal of this work is to mitigate such analysis issues and arrive at a more efficient and scalable analysis model.

6.2.2   Outline of Solution

Improving the performance of our CPN analysis method required the evaluation of our existing results to identify how the state space generation worked. The state space of CPN is a tree of CPN *markings*, where each marking is a data structure with specific values representing the tokens in all its places. So, our goal is to reduce the number of markings accumulated in the CPN, i.e., the number of distinct states of interest. This required us to evaluate our representation of time. Using time as a fixed-step monotonically increasing entity means that the CPN place managing time would always contain a new *clock token*, therefore forcing the CPN marking to become a part of the state space.

To alleviate this issue, we modeled time as a dynamically changing variable, where the

changes are strategically forced *time jumps* instead of a strictly incrementally increasing clock value. This is similar to a discrete event scheduling model but the next (closest) interesting event needs to be calculated based on the current state of the system. The states of all timers are used to calculate the next timer expiry. The states of all executing threads is evaluated to calculate the earliest next timestamp for thread preemption. Similarly, the state of all executing operations is analyzed to identify the earliest next enqueue onto the component message queue. When temporal partitioning is enabled, the next partition switching timestamp is also considered in effectively calculating the minimum amount of time by which the analysis clock on each node needs to be progressed to ensure that time is efficiently managed while ensuring that no interesting event is *skipped* because of the progression.

Similarly, our data structure representation for distributed deployments, i.e., using unordered token sets instead of ordered lists, enabled our earlier CPN models to nondeterministically choose one of the various distributed nodes to execute, generating a exponentially increasing tree of execution orders. Once we moved to representing our distributed hardware nodes as a list, the execution engine iteratively executing the analysis on each node in the list, leading to one execution order instead of a tree. Using a list has the same expressive power as using a set and does not constrain any resultant behavior. This is because the analysis model does not care about the order in which the hardware nodes are processed; simply the order (in time) in which the events and interactions take place. So, regardless of whether a set or a list is used, the same resultant behavior(s) is observed. By using lists instead of sets, the analysis model simply avoids needless non-determinism in the arc bindings when transitions fire.

Such issues are resolved with our analysis improvements, reported in [49]. We modified the timing analysis model to allow dynamic time progression, i.e., the clocks (one for each computing node) in the CPN model do not progress at a constant rate but instead experience *time jumps* to the next interesting time step e.g. next timer expiry, end of partition, next

61

scheduling preemption point, or next remote interaction. This makes the model execution progress at a much higher rate and reduces the overall number of states being recorded in the state space.

We also adjusted our modeling concepts when describing distributed deployments. We experienced needless state space explosions as a consequence of using CPN semantics when modeling distributed computers. If the computers are modeled as an aggregate of independent CPN tokens, then the CPN transition that progresses the execution in each computer is independent, leading to a potential $C!$ different orders for C computers. For instance, 4 distributed computers leads to 24 possible execution orders displayed by the transition responsible for *picking* the next computer to evaluate and progress. We alleviate this issue by assuming that all computers in a distributed scenarios have synchronized clocks and execute simultaneously leading to a synchronous progress. In CPN, this is done by maintaining the state of each computer in a *list* instead of an unordered aggregate. In practice this can be achieved by using the Precision Time Protocol (PTP) [21] to synchronize clocks throughout the computer network. The time protocol synchronizes all clocks within a network by adjusting clocks to the *highest quality clock*. The Best Master Clock (BMC) algorithm determines which clock is the highest quality clock within the network. The BMC then synchronizes all other clocks (slave clocks). This ensures that two computers in a network have synchronized clocks when an assembly of components is executed on the pair. Once the clocks on the computers are time-synchronized, the applications are started in parallel across the distributed computers. Thus, two periodic timers in two components executing on these two synchronized computers will trigger the respective components at approximately the same time instant.

6.2.3 Handling Time

The CPN-based analysis consists of executing a simulation of the model and constructing a state space data structure for the system (for a finite horizon), and then performing

queries on this data structure. This is automated by CPN Tools. The first improvement over the basic CPN approach is in how we handle time. Although it is true that CPN and similar extensions to Petri Nets such as Timed Petri Nets inherently have modeling concepts for simulation time, we explicitly model time as an integer-valued *clock* color token in CPN. There are several reasons for this choice.

Firstly, this is an extension to our previous arguments about choosing Colored Petri Nets. Modeling the OS scheduler clock as a colored token allows for extensions to its data structure such as (1) intermediate time stamps and internal state variables, and (2) adding temporal partitioning schemes like the (time-partitioned) ARINC-653 [7] scheduling model (Figure 26).

```
1`[{clock_node="Sat1", clock_value=0,
    schedule = [{part_name="Part1", exec_t=0, dur=20, pr=40,  off=0},
                {part_name="Part2", exec_t=0, dur=20, pr=40, off=20},
                {part_name="Part3", exec_t=0, dur=20, pr=40, off=40}]}]
```

Figure 26: A Clock Token with Temporal Partitioning

These extended data structure fields can be more easily manipulated and used by the model transitions during state changes, allowing for richer modeling concepts that would not be easily attainable using token representations provided by Timed Petri Nets. The ability to pack colored tokens with rich data structures also reduces the total number of colors required by the complete model. This quantitative metric directly influences the reduced size of the resultant state space. The downside of this approach to modeling is that we have to choose a time quantum. But in practical systems this is usually not a problem, as the low-level scheduling decisions are taken by an OS scheduler based on a time scale with a finite resolution.

Secondly, modeling time as a token allows for smarter time progression schemes that can be applied to control the pace of simulation. If we did not have such control over time,

the number of states recorded for this color token would combinatorially explode and itself contribute to a large state space. In order to manage this complexity, we have devised some appropriate *time jumps* in specific simulation scenarios.

If the rate at which time progresses does not change, then for a 1 msec time resolution, S seconds of activity will generate a state space of size: $SS_{size} = \sum_{i=1}^{S*1000} TF_{t_i}$ where TF_{t_i} is the number of state-changing CPN transition firings between t_i and t_{i+1}. This large state space includes intervals of time where there is no thread activity to analyze either due to lack of operation requests, lack of ready threads for scheduling, or due to temporal partitioning. During such idle periods, it is prudent to allow the analysis engine to *fast-forward* time either to (1) the next node-specific clock tick, (2) the next global timer expiry event, or (3) the next activation of the node-specific temporal partition (whichever is earliest and most relevant). This ensures that the generated state space tree is devoid of nodes where there is no thread activity.
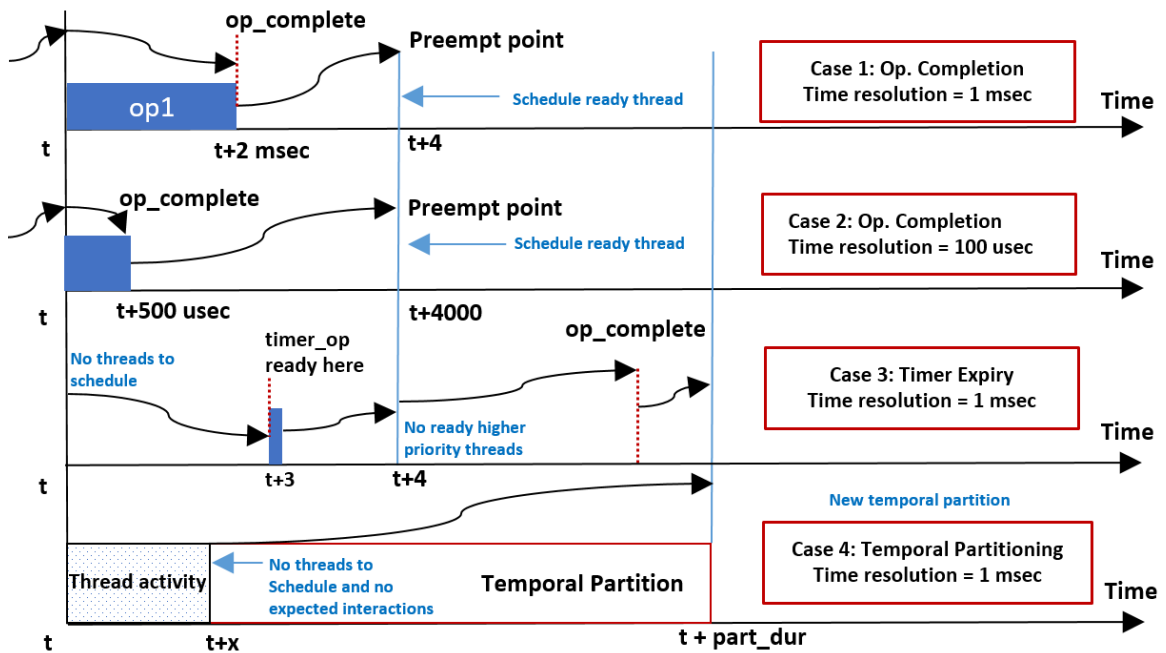

Figure 27: Dynamic Time Progression

Figure 27 illustrates these time jumps using 4 scenarios. Assuming the scheduler clock ticks every 4 msec, Case 1 shows how time progression is handled when an operation completes 2 msec into its thread execution. At time t, the model identifies the duration of time left for an operation to complete. If this duration is earlier than the next preemption point, then there is no need to progress time in $1msec$ increments as no thread can preempt this currently running thread till time $t + 4msec$. Therefore, the *clock_value* in Figure 26 progresses to time $t + 2msec$, where the model handles the implications of the completed operation. This includes possibly new interactions and operation requests triggered in other components. Then, time is forced to progress to the next preemption point where a new candidate thread is scheduled. This same scenario is illustrated in Case 2 when the time resolution is increased to 100 usec instead of 1 msec. Notice that the number of steps taken to reach the preemption point are the same, showing how the state space does not have to explode simply because the time resolution is increased. Case 3 illustrates the scenario where at time t, the scheduler has no ready threads to schedule since there are no pending operation requests but at time $t + 3msec$, a component timer expires, triggering an operation into execution. Since timers are maintained in a global list, each time the *Progress_Time* transition checks its firing conditions, it checks all possible timers that can expiry before the next preemption point. So, at time t when no threads are scheduled, the model immediately jumps to time $t + 3msec$. This scenario also shows that if the triggered operation does not complete before the preemption point *and* there are no other ready threads or timer expiries that can be scheduled, the clock value jumps to the operation completion. It must be noted here that this case is valid only because the DREMS architecture we have considered uses a non-preemptive operation scheduling scheme. Lastly, Case 4 shows time jumps working with temporal partitioning. At some time $t + x$, the model realizes the absence of ready threads and does not foresee any interaction requests from other components, then it safely jumps to the end of the partition without stepping forward in 1 msec increments. This time progression directly shows how the state space of the system execution reduces while

65

still preserving the expected execution order, justifying our choice of modeling time as a colored token using CPN.

6.2.4 Distributed Deployment

The second structural change to the analysis model is in how distributed deployments are modeled and simulated. Early designs on modeling and analysis of distributed application deployments [51] included a unique token per CPN place for each hardware node in the scenario. Since the individual *node* tokens are independent and unordered, there is non-determinism in the transition bindings when choosing a hardware node to schedule threads in. For instance, if there are 2 hardware nodes in the deployment with ready threads on both nodes, then either node can be chosen first for scheduling threads leading to two possible variations of the model execution trace. Therefore the generated state space would exponentially grow for each new hardware node. In order to reduce this state space and improve the search efficiency, we have merged hardware node tokens into a single *list* of tokens instead of a unassociated grouping of individual node tokens. This approach is inspired by the symmetry method for state space reduction [46].

Figure 28 illustrates this structural reduction. Consider a distributed deployment scenario with an instance of a DREMS application deployed on each hardware node, Sat1 through Sat6. Components *Comp1* and *Comp2* are triggered by timers, eventually leading to the execution of component operations (modeled as shown in Figure 17). If all the timer tokens in the system were modeled individually, the transition *Timer_Expiry* would non-deterministically choose one of the two timer tokens that are ready to expire at *t=0*. However, if the timers are maintained as a single list, then this transition (1) consumes the entire list, (2) identifies all timers that are ready to expire, (3) evaluates the timer expiration function on all ready timers, (4) propagates the output *operation* tokens to the relevant component message queues in a single firing. This greatly reduces the tree of possible transition firings and therefore the resultant state space. Also, if there is no non-determinism in

66
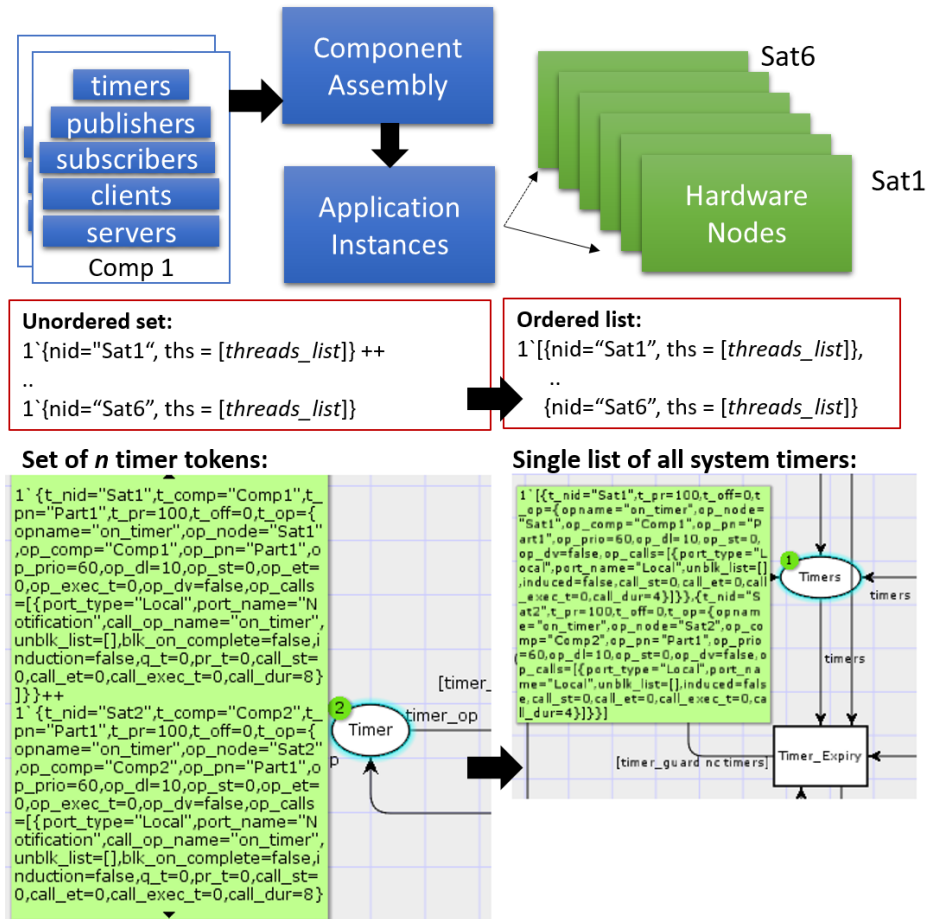
Figure 28: Structural Reductions in CPN

the entire system, i.e., there is a distinct ordering of thread execution, then this model can be scaled up with instantiating the application on new hardware nodes with no increase in state space size. This is because all of the relevant tokens on all nodes are maintained as a single list that is completely handled by a single transition firing.

An important implication of the above structural reduction is that the simulation of the entire system now progresses in synchronous steps. This means that at time 0, all the timers in all hardware nodes that are ready to expire will expire in a single step. Following this, all operations in all component message queues of all these nodes are evaluated together and appropriate component executor threads are scheduled together. When these threads

execute, time progresses as described in Section 6.2.3, moving forward by the minimum amount of time that can be fast-forwarded.

6.3 Scalability: Investigating Advanced State Space Analysis Methods

The main disadvantage of using state spaces for analysis is the state explosion problem: even relatively small systems may have an enormous number of reachable states, and this is a serious limitation for the use of state space methods in analysis of real-life systems. This has led to the development of many different reduction methods for alleviating the state explosion problem. Examples of such reduction techniques include partial order reduction methods [69, 85, 94], and the symmetry method [41]. Reduction methods represent the full state space in a compact, condensed form or represent only a subset of the full state space. The reduction is done such that the answer to the verification questions can still be determined from the reduced state space.

The scalability of the DREMS timing analysis CPN has been improved using the symmetry method, as described in the previous section. Instead of reinventing any state space reduction methods, the ASAP [91] analysis platform is used to effectively apply existing advanced state space analysis methods on the DREMS analysis model with no additional changes. In ASAP, the CPN analysis model is imported as such and integrated into a *verification project*. ASAP provides a platform on which to choose a variety of analysis parameters e.g. the type of graph traversal (depth-first, breadth-first etc.) and any analysis technique e.g. sweepline method as relevant.

Using the CPN Tools' built-in state space analysis tool, a bounded state space was generated reaching up-to 10 hyperperiods of component thread activity on a 100-component example. This bounded generation took 36 minutes on a typical laptop. The goal with using ASAP is to evaluate the effectiveness and utility of advanced state space reduction techniques that can be readily applied on any Colored Petri net, such as the DREMS analysis model. With no changes to the model, the model is analyzed in ASAP for system-wide

68

Table 1: Scalability Testing with CPNTools and ASAP – A bounded state space covering 10 hyperperiods of component interactions is generated

Tool	Test Case	Number of Embedded Nodes	Temporal Partitions per Node	Threads per Partition	Size of State Space	Generation Time
CPNTools	1	**5**	2	1	180	0.981 s
CPNTools	2	**2**	5	5	124,469	846 s
CPNTools	3	5	5	4	485,552	2190 s
ASAP	3	5	5	4	144,861	495.147 s

deadlocks in under 10 minutes. Table 1 summarizes these results. It must be noted that the size of the state space is reduced partly due to temporal partitioning itself. As described using Figure 22, the branching nature of the state space graph is dependent on (among other factors) the number of "ready" equal-priority component executor threads. With temporal partitioning, this number reduces down to a fraction of the total number of components. Without temporal partitioning, if all 100 threads are theoretically ready to execute, then the size of the state space would be in order of millions of nodes and the state explosion would make any analysis intractable. So, it is observed that though tools like ASAP help alleviate the state explosion and make state space generation occur in much faster rates, the DREMS timing analysis model is still affected by this explosion in very large-scale systems with potentially thousands of components.

CHAPTER VII

EXPERIMENTAL EVALUATION

Experimentally validating our timing analysis results is an important and necessary requirement. In order to obtain any level of confidence in our CPN-based work, the system design model needs to be completed implemented, and deployed on the target hardware platform. We have constructed a testbed [47] to simulate and analyze resilient cyber-physical systems consisting of 32 Beaglebone Black development boards [1]. We have chosen the light-weight ROS [71] middleware layer and implemented our ROSMOD Component model [48] on top of it. This component model provides the same execution semantics and interaction patterns as our DREMS component model [66]. Our goal with this work was to (1) establish a set of distributed component-based applications, (2) translate this design model to our CPN analysis model, (3) deploy these applications on our testbed and accurately measure operation execution times, and finally (4) perform state space analysis on the generated CPN model to check for conservative results, compared against the real system execution.

7.1 Challenges

Experimental validation requires that online measurements of the real-time system match with the design-time timing analysis results in a way that the timing analysis results are always close but conservative. If the timing analysis results predict a deadline violation, this does not necessarily mean that the real system will violate deadlines but if the timing analysis and verification guarantees the lack of deadline violations, then the real system should agree with this prediction. The design-time timing analysis primarily uses bounded state space analysis for such predictions. The analysis is bounded by necessity i.e. to tackle the state space explosion problem. Obtaining confidence from the timing analysis

results depends entirely on the behavior of the system and the applied bounds. Recall that DREMS components are dormant by nature and need to be triggered either by timers or through interaction ports. Component-based applications using DREMS typically exhibit some periodic behavior, i.e. sequences of interactions between components that repeats periodically. Each interaction sequence could involve a set of distributed components or assemblies. In such scenarios, design-time analysis bounds the state space generation to some constant multiple of the overall period of the application. The overall period here is some duration of time after which all of the sequences of interactions in the application have repetition. By analyzing some reasonably large multiple of the application period, the state space analysis both generates and searches a sufficiently large set of states of the system, i.e. execution behaviors. Assuming the execution correctness of the timing analysis model and sufficiently conservative worst-case estimates of execution times, a complete absence of timing anomalies in this bounded state space is typically a good indication of a safely executing system. There are various ways to obtain the WCET values for individual operational steps but the easiest approach is to execute the design on our testbed and make accurate measurements.

7.2 Measurement and Instrumentation of Component Operations

As described in Section 5.3, the business logic of component operations is a set of computational steps. The ordering of execution of these steps, and also the semantics of each step directly affects the worst-case execution time of the block of code. Thus, in order to integrate the worst-case aspects of a component operation when modeling the business logic, these aspects need to be measured.

WCET of component operational steps needs to be measured by having each component operation execute at highest priority with no other component threads intervening this

process, i.e., each operation must be executed stand-alone on the CPU with as little inter-vention as possible from other components. If a component operation requires an inter-action, e.g., a client-server or publish-subscribe interaction, then the source or destination of this "connection" must execute on a different CPU. That way, the CPU of the opera-tion being measured isn't affected by the other components/operations that are part of the interaction.

Secondly, this experiment must be repeated multiple times with different inputs (if any), including the worst-case input. This is necessary to ensure complete coverage of the busi-ness logic code so that the worst-case execution time of the operation is accurately and consistently measured. In the past, researchers have applied various test case generation methods including (1) translating Statecharts to Finite State Machines (FSM) and using tools like Condado [79], or (2) using model checking and symbolic execution with the Java PathFinder model checker [86] or (3) WISE [19] (Worst-case Inputs from Symbolic Execution). WISE, for example, uses exhaustive test generation for small input sizes and generalizes the result of executing the program on those inputs into an "input generator."

The business logic code of the operation is instrumented with logging statements, i.e., function calls part of a logging infrastructure that present as little overhead as possible. Our logging framework used a circular buffer or ring buffer implementation of fixed size and ensured that the logged strings are written to disk only when the destructor of the logging class is called. That way, during the execution of the component operation, the delays caused by the act of logging is kept to a minimum. Log statements are injected into different parts of the code, e.g., right before a component interaction, or a looping statement.

Recall from earlier discussion that conditional statements are not modeled in our busi-ness logic. Modeling data-driven behavior, e.g., if-else statements, is avoided to keep the state space as reduced as possible. However, this presents a challenge; if there is an infre-quently executed (e.g. twice) "if" statement that takes 5 s, and a frequently executed (e.g.

300 times) "else" statement that takes 2 ms, which WCET should the analysis consider? Should the "else" statement be considered since it happens significantly more often? Here, we have decided to stick to the worst-case behavior, i.e., always choosing the "if" case. The problem with this choice is that this will lead to a gross over-estimation of the operation's behavior and this exposes the weakness in our busines logic model. Regardless, for conditional statements in the actual business logic, logging statements are added before the start of the first conditional statement and after the end of the last conditional statement. The execution time for a set of conditional statements, i.e., multiple if-else cases that are written in a sequence by the programmer, are evaluated together and the worst-case is identified.

The worst-case measurement for the operational steps across multiple runs of the experiment gives us a *pure execution times* of the various code blocks in an operation. This measurement is called pure because it is made with as little interference as possible from other component operations. These measurements are used to construct the business logic models in the system model. From here, the analysis CPN is generated and the measurements are injected as tokens into CPN places, e.g., timer tokens, interaction tokens, etc. Once a complete CPN is generated, bounded state space analysis is performed. The bound on the state space generation is represented in terms of some number of clock ticks, calculated as some large multiple of the lowest frequency event in the system, e.g., is the lowest frequency event is a timer that fires every second, then the state space is generated to cover, e.g., 100 seconds of activity, covering 100 timer expiries. This bound should ideally be derived mathematically by considering all periodic events in the system and calculating the minimum amount of time for which the system must be observed to either catch timing violations or declare that the system is safe. In our analysis, these bounds are chosen simply based on experience, i.e., a duration of time deemed to be "sufficiently large", e.g., at least 10 hyperperiods of a temporal partition schedule.

The final step of the analysis process is to compare the analysis results for the composed system, i.e., all software components executing as planned in the final deployment,

against the exact same deployment on a real testbed. Here, the measurement of interest is the worst-case response time of all component operations, i.e., for each component operation, the time taken from when the operation was requested in the message queue till when the component executor thread completes execution of the operation business logic. The execution of multiple operations can be grouped to calculate worst-case trigger to response times. Similar to the pure execution time measurements, the testbed deployment of the composed system must be repeated multiple times with various inputs in order to ensure that the worst-case is truly captured. In order to perform such evaluation, we have constructed a testbed called the RCPS testbed. This is described below.

7.3 Resilient Cyber-Physical Systems (RCPS) Testbed

7.3.1 Architecture

The RCPS testbed, as shown in Figure 29 consists of 32 *RCPS nodes*, each of which is a Beaglebone Black (BBB) [1] development board. For the subset of CPS we are interested in, the behavior of the CPS can be much more precisely emulated with these boards compared to running the applications inside of a standalone simulation. For example, NASA's CubeSat Launch Initiative (CSLI) [4] provides opportunities for nanosatellites to be deployed into space for research. CubeSats are small (4-inch cubes) satellites running low-power embedded boards and being prepared for interplanetary missions [5] to Mars. A distributed set of CubeSats can be easily tested with this architecture if it can be integrated with a high-fidelity space flight simulator; each RCPS node would run the embedded software of a single CubeSat.

The Gigabit Ethernet port of each BBB is connected to a *Communication Network* switch. This is a programmable OpenFlow [58] switch, allowing users to program the flowtable of the switch to control the routes that packets follow and completely configure the full network and subnets required for their emulated deployment. Furthermore, the configurability of the communications network enables per-link or per-flow bandwidth
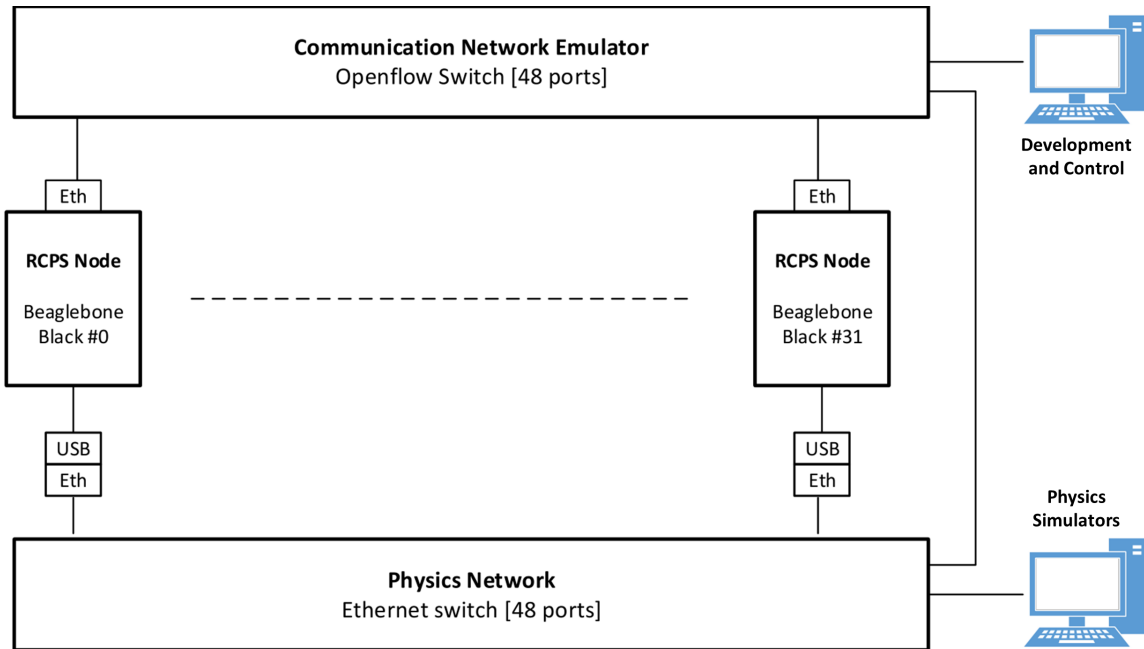
Figure 29: Testbed Architecture

throttling, enabling precise network emulation. The primary *Development and Control* machine, running our software development tools, communicates with the BBBs using this network. After software applications are deployed on this testbed, the characteristics of the real CPS network can be enforced on the application network traffic. Therefore, this network emulates the physical network which a distributed CPS would experience on deployment.

Each RCPS node is also connected to a *Physics Network* using a 10/100 USB-to-Ethernet adapter, since the BBBs only have one gigabit ethernet port. This network is connected to a *Physics Simulation Machine* running Cyber-Physical Systems simulations. This network provides the infrastructure necessary to emulate CPS sensing and actuation in the loop, allowing application software to periodically receive sensor data and open interfaces to output actuator commands to the simulation.

The Physics Simulation Machine closes the interaction loop for the testbed nodes, allowing the physical dynamics of the RCPS nodes to be simulated in the environment in

which it would be deployed, *e.g.* satellites' orbital mechanics and interactions can be simulated for a satellite cluster.

7.4   ROSMOD Software Infrastructure

We execute component-based software applications on the RCPS testbed using a model-driven development tool called ROSMOD [48]. ROSMOD is an integrated development environment for rapid prototyping component-based software for the Robot Operating System (ROS) middleware. ROSMOD is well suited for the design, development and deployment of large-scale distributed applications on embedded devices. ROSMOD includes a modeling language, a graphical user interface, code generators, and a deployment infrastructure. The utility of ROSMOD has been demonstrated in the past with a real-world case study: an Autonomous Ground Support Equipment (AGSE) robot that was designed and prototyped using ROSMOD for the NASA Student Launch competition, 2014âĂŞ2015.

7.4.1   ROSMOD vs. DREMS

ROSMOD includes an implementation of the DREMS component model. However, there are some minor differences between ROSMOD and the original DREMS. Primarily, the middleware layer in DREMS, i.e., the layer responsible for providing the various supported communication patterns has been swapped out for the simpler, light-weight ROS middleware. Among other factors, this reduces the effort required on the developer's part to rapid prototype a distributed component-based application. This switch to ROS also shows the versatility of the DREMS component model. Secondly, the ROSMOD infrastructure does not implement any temporal partitioning like DREMS. The goal here is to also show that the CPN analysis model is flexible enough to model and analyze systems without temporal partitioning constraints.

7.4.2 ROSMOD Modeling Language

To enable the design, development, and testing of software on distributed CPS, we have developed a modeling language specific to the domain of distributed CPS which utilize ROS, the ROSMOD Modeling Language (RML). Figure 30 shows the metamodel for this language using GME [52] notation; the GME-based metamodel figure is very similar to a traditional UML class diagram with some minor differences in notation. RML captures all the relevant aspects of the sofware, the system (hardware and network), and the deployment which specifies how the software will be executed on the selected system. Using ROSMOD, developers can create models which contain instances of the objects defined in RML. This approach of using a domain specific modeling language to define the semantics of the models allows us to check and enforce the models for correctness. Furthermore, this approach allows us to develop generic utilities or extensions, called *plugins* [56] which can act on any models created using ROSMOD, for instance generating and compiling the software automatically or automatically deploying and managing the software on the defined system. The rest of this section goes into the specific parts of the modeling language, called the metamodel, and how they define the entities in a ROSMOD Model.
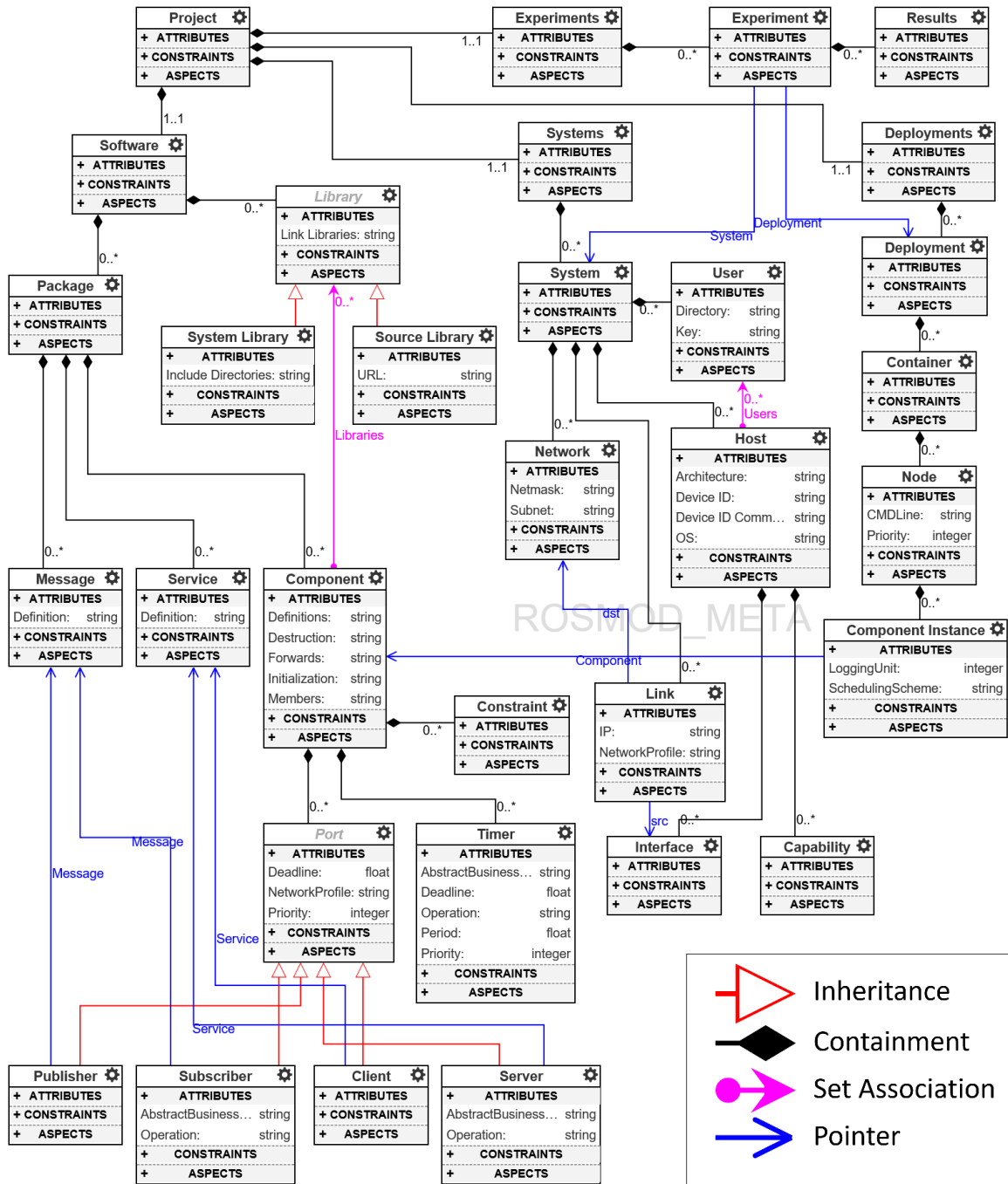
Figure 30: ROSMOD Metamodel

The top-level entity of RML is a *Project*, which is shown in the upper left of Figure 30. The language supports a variety of modeling concepts that address structural and behavioral aspects for distributed embedded platforms. ROSMOD users can create models of software workspaces, required software libraries, embedded devices, network topologies, component constraints and hardware capabilities. The language also supports code development, particularly with regards to port interface implementations i.e. the execution code for operations owned and triggered by communication ports or local timers. Below, we describe in detail the various aspects of this metamodel and how these concepts are integral to developing distributed CPS and rapid prototyping needs.

7.4.3 Motivation for ROSMOD Software Model

The goal of the ROSMOD software model is to provide a language to precisely model the application software. When using a DREMS-style component model, the software is primarily a collection of components, where each component is defined by its ports and timers. Building a precise model of the software has various benefits. Firstly, applying model-driven development techniques enables reuse of previously defined components i.e. a single component can be instantiated or copied or modified as required and executed on the runtime system. Secondly, the development time of the application can be reduced significantly as a large part of the runtime code can be fully generated based on templates. Lastly, a clear model of the software provides a canvas for design-time analysis. If the structural aspects of the software are captured in the model of the component assembly, and the behavioral properties of the components are encoded in the attributes of its ports and timers, then using translation rules, a design-time analysis model can be fully generated.

7.4.4 Software Model

The *Software* class in Figure 30 models a software workspace. A workspace, following ROS terminology, is a collection of applications that are developed and compiled together

into binaries. Thus, each Software class can contain ROS applications, called *Packages*, and *Libraries* required for the applications. Packages consist of *Messages*, *Services* and *Components*. Components contain a set of pointers to Libraries to establish dependence e.g. an *ImageProcessor* component *requires* OpenCV, an open-source computer vision library. Libraries are of two types: Source libraries and System libraries. Source libraries are standalone archives that can be retrieved, extracted and integrated into the software build system with no additional changes. System libraries are assumptions made by a software developer regarding the libraries pre-installed in the system. Here, system refers to the embedded device on which the component is intended to execute.

Messages represent ROS message types used by publisher and subscriber ports for topic-based communication. Similarly, *Services* describe the ROS peer-to-peer request-reply interaction pattern. Each service is characterized by a pair of messages, *request* and *response*. A client entity can call a service by sending a request message and awaiting a response. This interaction is presented to the user as a remote procedure call. Each ROSMOD component contains a finite set of communication ports. These ports refer to messages and services to concretize the communication interface. Components can also contain *Timers* for time-triggered operation e.g. periodically sampling inertial measurement sensors while operating an unmanned aerial vehicle (UAV).

7.4.5 Motivation for ROSMOD System Model

The goal of the ROSMOD system model is to provide a language to precisely model the network of computers capable of executing applications defined in the software model. The system model is necessary for both compilation and deployment. The software defined in the software model, and therefore the generated source code must be compiled down to a binary for runtime execution. An accurate model of the available runtime system provides necessary information for cross-compilation requirements. The deployment

framework can use the system model to find a suitable candidate device onto which the application processes are deployed. As shown in Figure 30, a deployment consists of a set of abstract containers, where each container is a set of processes. The deployment infrastructure maps each container to one available RCPS node based on availability; regardless of which RCPS node a container maps to, the set of processes contained by that container are always deployed together. To automate this process, the system model must capture fine-grained details about each available device, including information such as the IP address, the user permissions, and means to access the device e.g. Secure Shell Protocol (SSH) [95]. As with the software model, this system model can be reused in all ROSMOD projects for a given hardware assembly, such as the RCPS tested.

7.4.6 System Model

A *System Model* completely describes the hardware architecture of a system onto which the software can be deployed. A ROSMOD Project contains one or more *Systems*. Each System contains one or more *Hosts*, one or more *Users*, one or more *Networks*, and one or more *Links*. A host can contain one or more *Network Interfaces*, which connect through a link to a network. On this link the host's interface is assigned an IP address, which matches the subnet and netmask specification of the network. Additionally, a host has a set of references to users, which define the user-name, home directory, and ssh-key location for that user. The host itself has attributes which determine what kind of processor architecture it has, e.g. *armv7l*, what operating system it is running, and lastly a combination of Device ID and Device ID Command which provide an additional means for specifying the type of host (and a way to determine it), for instance specifying the difference between a BeagleBone Black and an NVIDIA Jetson TK1 which both have armv7l architecture but can be separated by looking at the model name in the device tree. Finally, a host may contain zero or more *Capabilities* to which the component constraints (described in the previous section) are mapped. The final relevant attribute is the *Network Profile* attribute of a link.

Using the network profile, which is specified as a time-series of bandwidth and latency values, we can configure the links of the network using the Linux TC to enforce time-varying bandwidth and latency. This network configuration is useful when running experiments on laboratory hardware for which the network is not representative of the deployed system's network.

7.4.7 Deployment Infrastructure

The workflow for software deployment is as shown Figure 31. After the user has generated and compiled the software model into binary executables, they can run an experiment that has valid deployment model and system model references. Every ROSMOD workspace is generated with an additional *node* package. This builds a generic node executable that can dynamically load libraries. When the software infrastructure generates and compiles the source code for the software model, the components are compiled into dynamically loadable libraries, one for each component definition along with a single executable corresponding to the generic node package. The first step the deployment infrastructure performs when running an experiment is generating the XML files which contain metadata about each ROS node modeled in the ROSMOD Deployment Model. This metadata includes the component instances in each node and the appropriate component libraries to be loaded. Based on the XML file supplied to the node executable, the node will behave as one of the ROS nodes in the deployment model. This allows for a reusable framework where a generic executable (1) loads an XML file, (2) identifies the component instances in the node, (3) finds the necessary component libraries to load and (4) spawns the executor threads bound to each component.
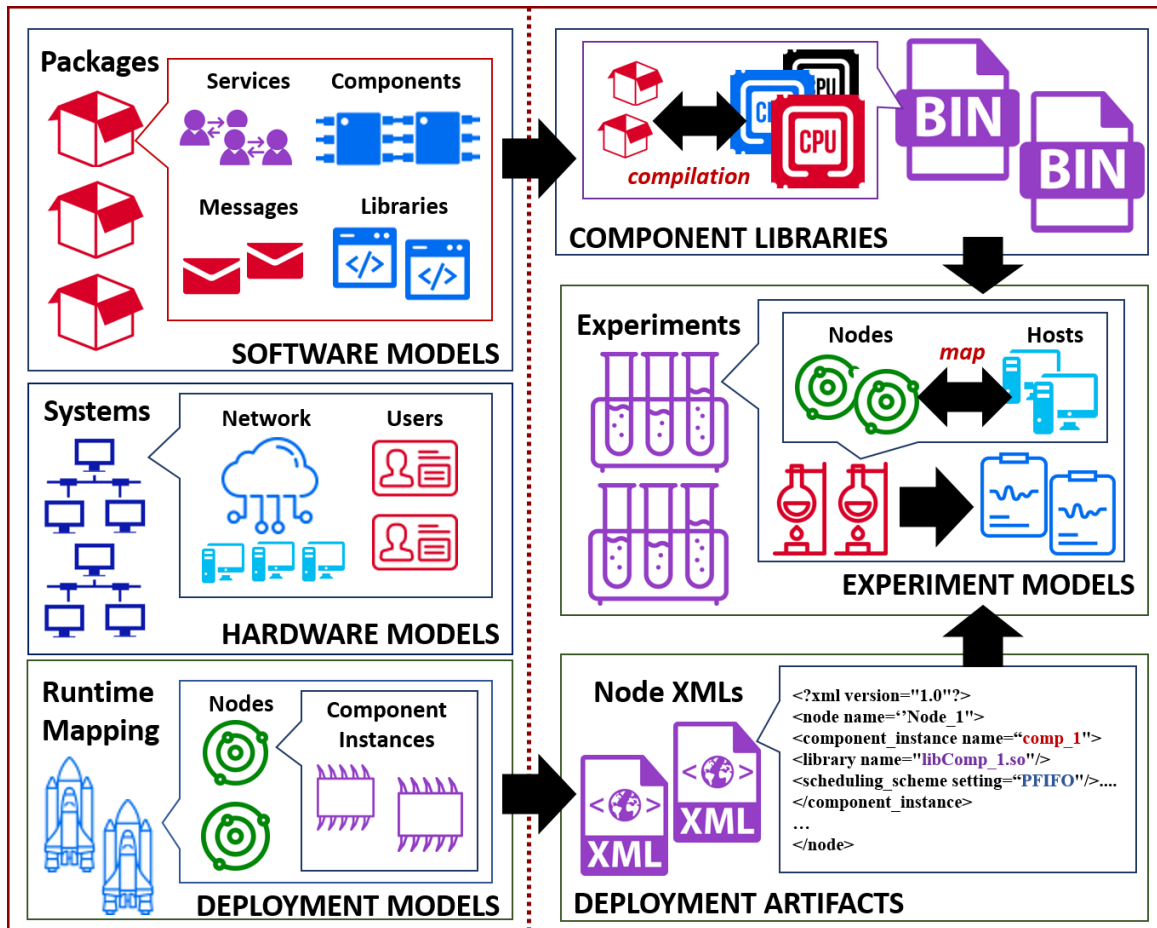
Figure 31: Software Deployment Workflow

7.5   Validation of Timing Analysis Results

Validation of the modeling and analysis requires strict evaluation of the execution traces generated by the state space analysis and comparing these traces against the traces observed in the experiments. The CPN model is validated if an execution trace, i.e., an order in which the component operations have executed, can be shown in the CPN generated state space. If so, the the state space accounts for the real behavior and this would show that the CPN model of the component behavior is consistent with the expectations of the user. If the generated state space does not contain the execution trace observed in the experiments, then this will show expose (1) errors in the CPN model of the behavior, or (2) insufficient non-determinism considerations, e.g., if the non-determinism does not correctly account

83

for all the possible thread, timer, or operation executions during a time period, then the analysis will be incorrect and unreliable.

Experimental validation should also demonstrate that online measurements of the real-time system match with the timing analysis results in a way that the timing analysis results are always close but conservative. The goal of the analysis is to obtain a fairly accurate estimation of the runtime behavior i.e. estimates of component timing behavior that isn't so conservative that the results are useless. If the real execution of a specific operation takes 100 milliseconds and the timing analysis predicts 107 milliseconds, then this is close e.g. less than 10% error, but conservative. On the other hand, if the design-time analysis predicts the execution time to be in the order of seconds or tens of seconds, then the analysis is conservative but grossly over-estimate. One of the biggest assumptions in our CPN work is the knowledge of worst-case execution times of the individual steps in the component operations. As detailed in Section 5.3, the business-logic modeling language captures the temporal behavior of component operations, especially WCET metrics for the different code blocks inside an operation. For example, consider a simple client-server example as shown in Figure 11. The client component is periodically triggered by an internal timer and executes a synchronous remote method invocation to a remote server component. The interaction here demands that the client component be blocked for the duration of time it takes the server to receive the operation, process its message queue, execute the relevant callback, and respond with output.

Note that in Figure 11, we only annotate isolated code blocks that take a fixed amount of execution time on a specific hardware architecture. These are the only measurements that we can reliably make with repeated testing and instrumentation. The client-side blocking delay is not measured because there are numerous factors responsible for this delay e.g. server's message queue state, scheduling non-determinism, network delays etc. In order to be able to predict this delay, we need to use state space analysis and search through the tree of possible executions to identify the worst-case blocking delay. This also means that our

84

CPN model must capture and account for such delay-causing factors. Lastly, in order to reliably obtain worst-case response times from the analysis, the analysis must account for all significant non-deterministic possibilities, e.g., timers expiring at the same time as other operation inductions, threads unblocking at the same time as other component interactions, etc.

The remainder of this section presents various primitive interaction patterns and assemblies that have been evaluated. The results are restricted to simple cases, though we have tested on medium-to-large scale examples spanning 25-30 computing nodes, and with up to a 100 components. The scalability of our model, however, is not within the scope of this paper as we have previously evaluated this metric [49]. As mentioned earlier, in all of our tests, we use the ROS [71] middleware and our ROSMOD [48] component model.

7.5.1   Note on the Beaglebone Black RCPS nodes

The Beaglebone Black embedded devices in the RCPS cluster presented an interesting challenge during experimentation. These devices are low-power embedded boards that are expected to run at 1 GHz CPU clock frequency. These devices, while running Ubuntu, are also influenced by a *CPU scaling governor*. This governor provides a mechanism for CPU throttling by setting the devices to run at different *modes*, e.g., power-saver, performance, on-demand etc., depending on the operational needs. These modes affect the CPU clock frequency ranges in which device operates, e.g., power-save limits the CPU frequency to 300 MHz, instead of 1 GHz, Similarly, the on-demand scaling mode throttles the CPU frequency to within the range of 300-600 MHz depending on the CPU demand. These variations are harmful to our experiments, as the execution times of the component operations will vary wildly depending on the clock frequency at the time of execution. In order to prevent this, not only is the scaling governor set to *performance* mode, but the minimum scaling frequency is also forcefully set to 1 GHz. This way, the clock frequency is forced to run at 1 GHz for 100% of the time regardless of the load on the CPU. Secondly, the

components threads are executed in real-time priority, with the real-time thread scheduling scheme set to SCHED_RR, i.e., fixed-priority scheduling with round-robin conflict resolution.

7.5.2   Understanding the CPN Analysis Plots

By performing state space analysis, we are analyzing a bounded tree of possible component behaviors. By identifying the worst-case execution trace in this tree, we're able to obtain a suitable conservative candidate execution that represents a possible behavior. Once this trace is identified, we plot the response time behavior of all components in this trace. This pattern is followed in all of the following plots. Figure 32 describes the analysis plots presented later in this chapter. Each subplot in this figure represents the execution of a component operation. The x axis of this plot represents the analysis time, and the y axis represents the response time of the operation. Each execution is shown as a rectangular pulse, the amplitude of which is the worst-case response time (WCRT) of the operation i.e. the time taken for the operation to complete (response) from when the executor thread was released for execution (trigger). The rising edge of the pulse represents the enqueue time stamp of the operation i.e. the time instant when a request for this operation was enqueued onto the component message queue. The falling edge of the pulse represents the completion time stamp i.e. the time instant when the component executor thread has completed execution of the operation and is ready to service the next request waiting in the queue. Since the response time of the operation is calculated from the enqueue time instant, the plot can have intersecting pulses, as shown in the second subplot. Here a new operation request is enqueued onto the message queue while an existing instance of the operation request is being executed by the component executor thread.
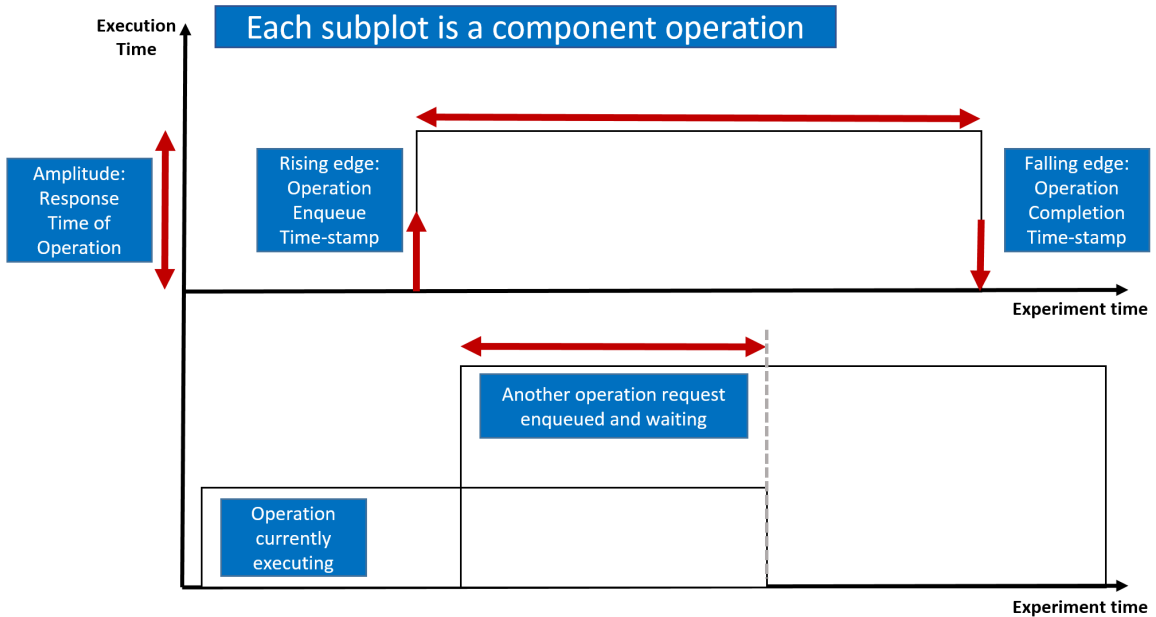
Figure 32: Interpreting Execution Time Plots

The following sub-sections present some use-cases and example deployments on which the CPN analysis has been tested. We begin testing with some primitive interaction patterns in order to ensure that the timing analysis model correctly executes as per the semantics of the interactions, e.g., blocking behavior in client-server interactions, PFIFO scheduling schemes on the component level, etc.

7.5.3 Client-Server Interactions

As shown in Figure 11, a simple client server example. A client component is periodically triggered into execution by a component timer (period = 5 s). The timer expiry causes an operation request to manifest in the client component message queue. The client component executor thread then executes this operation, i.e., executes the computational steps in the operation's callback, calls a service exposed by the server component, and blocks until the server responds. At this point, a client request manifests on the server component

message queue. The server component executor thread picks this operation request, executes the server callback and responds to the client, essentially unblocking the client. This behavior repeats periodically.
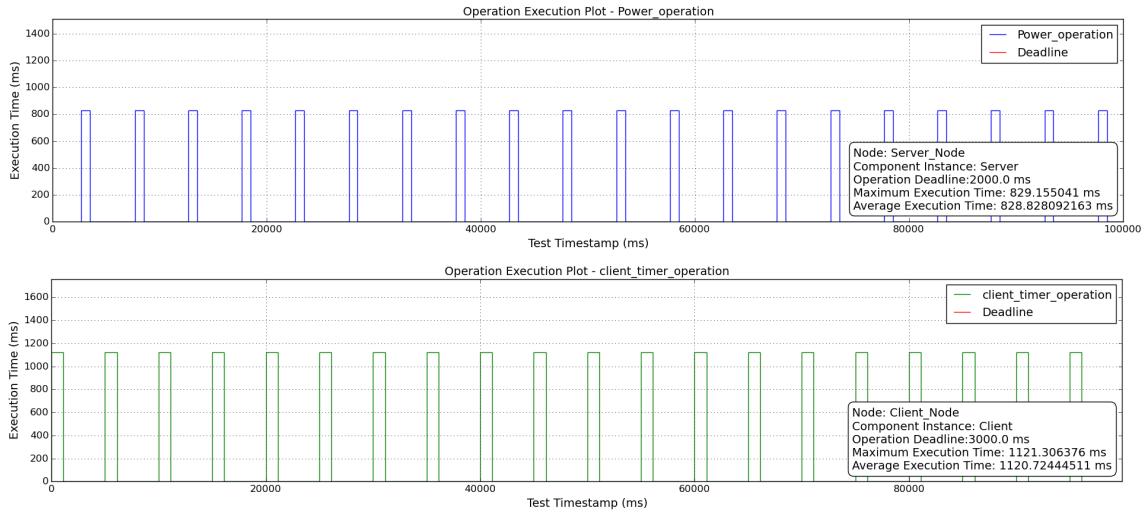

Figure 33: Experimental Observation: Client-Server Interactions

As mentioned in the experiments workflow, each component operation is executed separately on the testbed to obtain the pure execution times, i.e., uninterrupted execution times of the operations. These numbers are plugged into business logic models to generate a complete, analysis-ready CPN. Once the CPN is ready, a composed system-level analysis can be performed. Here, all the components in the assembly, i.e., the client and the server are deployed together in the RCPS testbed and the operation response times are accurately measured with as little overhead as possible. The deployments are repeated at least 10-15 times, with each run spanning some large multiple of the lowest frequency event, e.g., 100 seconds of activity is recorded when the triggering timer fires every 5 seconds. We believe this is sufficient periods of activity, especially for such a simple example, to obtain a distribution of execution times. Once all the deployments are finished, the logs are collected and the worst-case response times for all operations are derived using a Python-based script. When the worst-case logs are identified, these logs are used as the candidate

worst-case execution and plotted. Figure 33 shows the worst-case behavior observed over all the experimental runs.

The following figure shows a histogram of the measurements; the plots are generated using a Python-based script using a 25-bin histogram configuration. As it can be seen here, there is very little difference between the average-case and worst-case behavior for both operations. This is because there is very little variability in the business logic and timer frequency is not so large as to affect the delays caused by the blocking interactions.
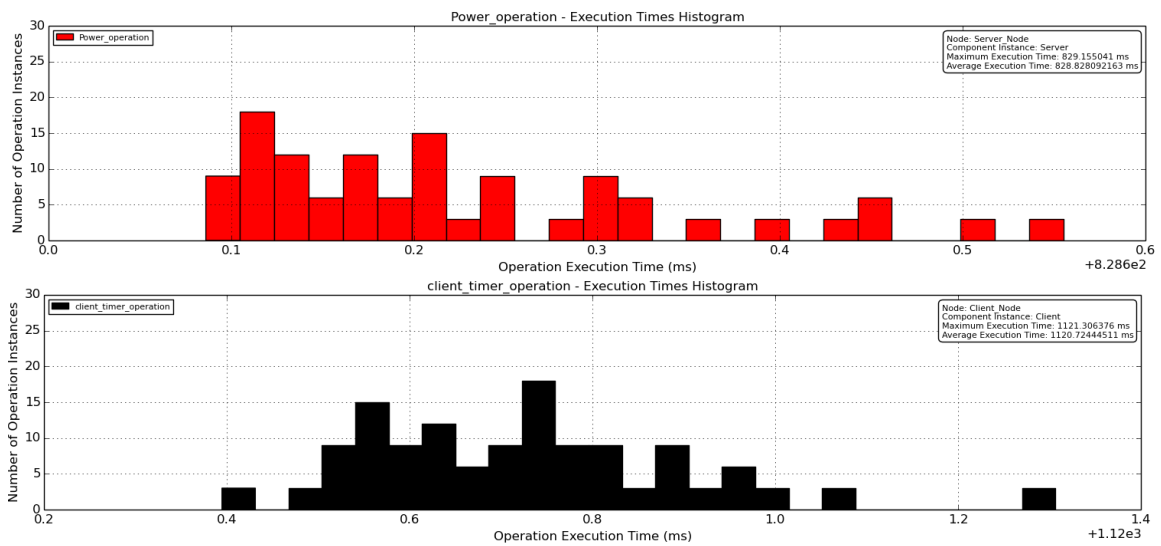

Figure 34: Histogram of Measurements for Client-Server Scenario

Using the generated analysis CPN, we perform bounded state space analysis covering about 100 seconds of thread activity on both client and server nodes. Searching through this state space, a list of operation response times are generated for both operations. The state space node containing the worst-case response time is identified for each operation and a backtrace is calculated using the state space analysis function interface. Then, this backtrace is written to a file and plotted. Figure 35 shows the execution time plot derived from our CPN. As expected, since there are no other interruptions on the server side, the server is able to promptly respond to the client.
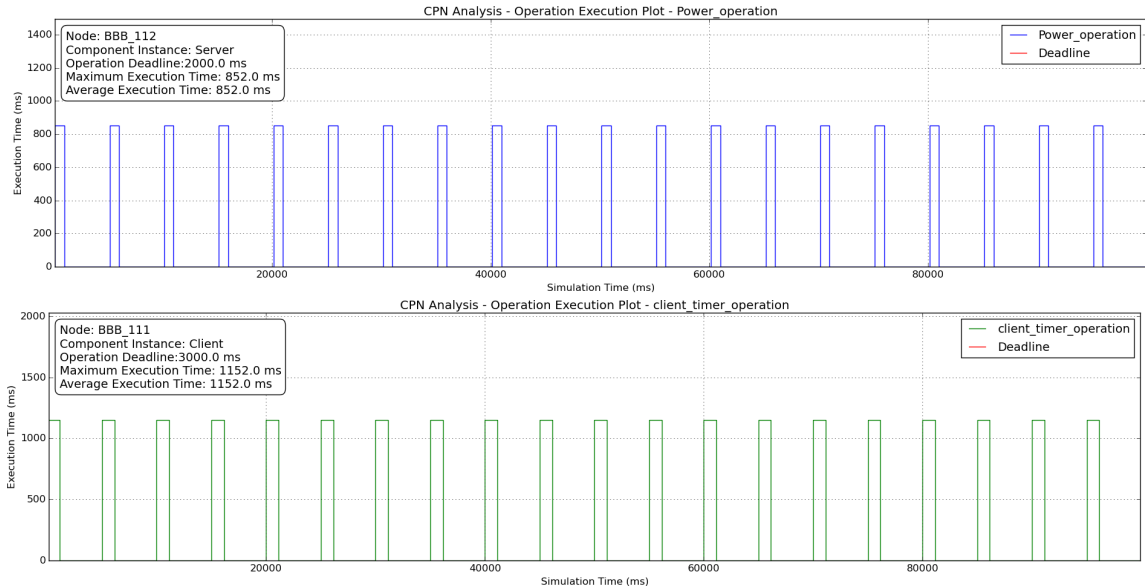
Figure 35: CPN Analysis Results: Client-Server Interactions

Table 2 shows a summary of these results. Similar tables are included as part of all other experiments in this section.

7.5.3.1 Bad Designs

The goal of our CPN timing analysis is to identify bad component designs, unacceptable execution times, response times etc. There are various ways in which we can accidentally design a poorly performing client-server interaction. In the above case, the server operation

Table 2: Client Server Example - Summary of Results

Operation Name	Component Name	Measured Response Time (μ, σ) (ms)	Measured WCRT (ms)	CPN Analysis WCRT (ms)	Deadline (ms)	Error (%)
Power Operation	Server	(828.82, 0.11)	829.15	852.0	2000	2.75
Client Timer Operation	Client	(1120.72, 0.03)	1121.30	1152.0	3000	2.71

90

takes 852 ms in its worst-case before responding to the client and unblocking the client executor thread. If instead, the server operation took 8.5 seconds, the client component will stay blocked for 10 times longer and the client timer expiries will not be serviced faster than the timer periods. This shows a simple use-case where the currently blocked client timer operation is starving subsequent timer expiries from being handled promptly. Figure 36 shows our CPN predictions after simply changing this server execution time.
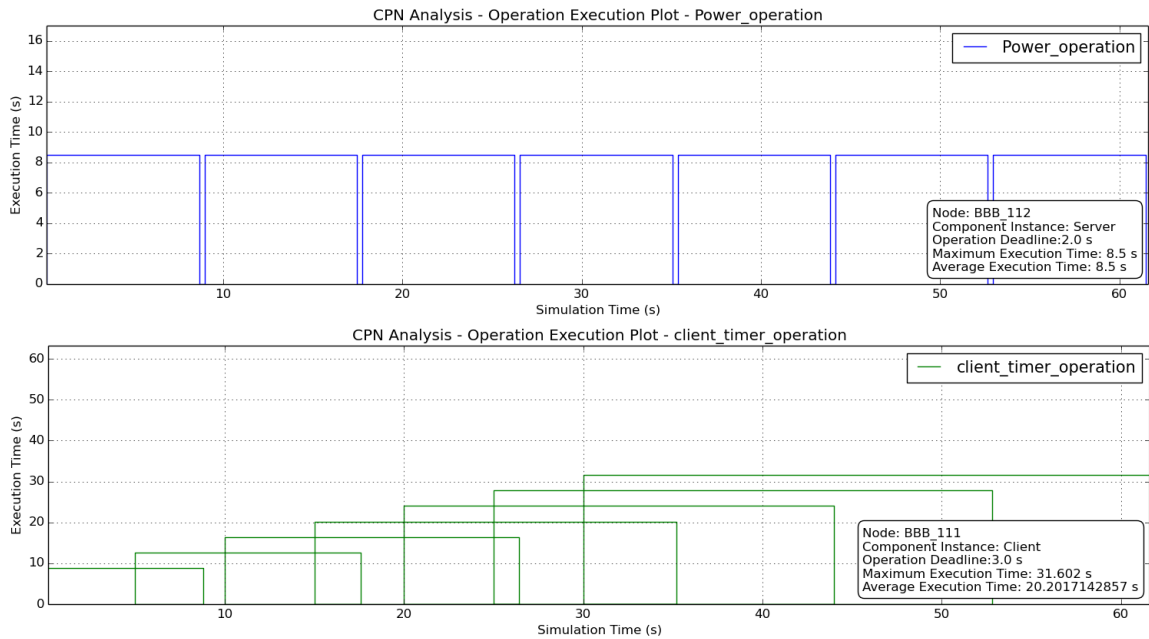

Figure 36: CPN Analysis Results: Client-Server Response Times in Bad Designs

The execution time of the each new client-side timer operation is worse than the previous since the operation is spending much longer waiting in the queue. Recall that the execution time of a component operation includes the waiting time in the message queue. Even with a bounded state space that spans just 1 minute, it is clear that the client component message queue size is monotonically increasing. This is a use case where a client component execution is affected by delays caused on a remote server. Each client-server

interaction delay will only be worsened when the server component has other operations to tend to aside from the client requests.

7.5.4 Publish-Subscribe Interactions

Similar to the earlier example, consider the ROSMOD publish-subscribe interaction. A publisher is periodically triggered by a timer when this component broadcasts a message on a topic. A subscribing component receives this message and performs some computation. In this case, the timer period is set to 2 seconds i.e. every 2 seconds, these two component interact via publish-subscribe messaging passing.

In order to enforce some variabilities in the execution, the publish-timer and subscriber operation business logic consists of algebraic computations performed in a loop. An upper limit for the number of iterations is assumed. At each invocation of this operation, a loop count somwewhere between 20% and 60% of the upper limit is randomly picked from a uniform distribution. The computations repeat for the chosen numer of iterations before the operation is marked as complete. We also ensure that any compiler optimization on this computation is disabled. Figure 37 shows experimental observations for this component assembly. Figure 38 shows the histogram of the measurements.
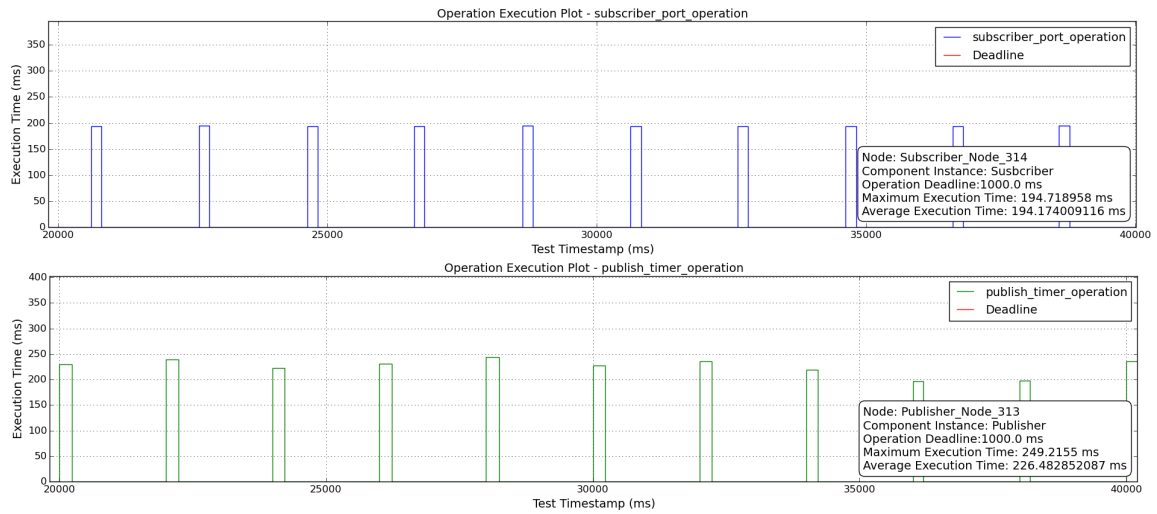
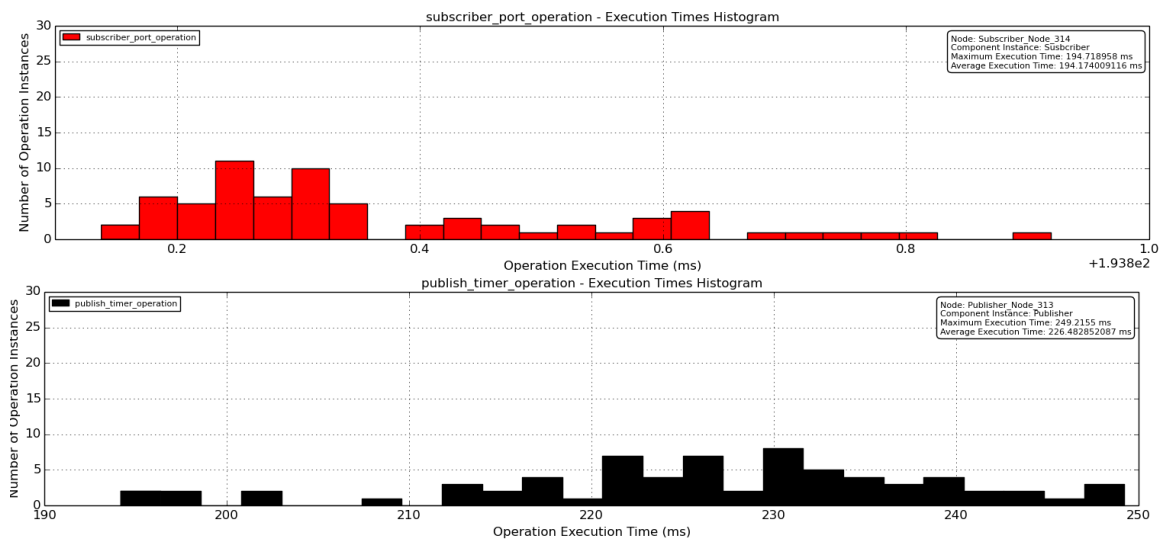**Figure 37: Experimental Observation: Publish-Subscribe Interactions**

Figure 38: Histogram of Measurements for Publish-Subcribe Scenario

93

Figure 39 shows our CPN analysis results. As evident, the CPN results closely matches and validate this sample. Table 3 summarizes the results. Since this is another simple use-case, there is not much non-determinism involved if the components are executing in separate devices. The execution trace, based on the design, follows a simple and deterministic ordering. Evaluating simple cases like this is important in order to ensure that the CPN correctly models and executes these types of interaction patterns. In the following examples, we also evaluate more integrated and complex scenarios.
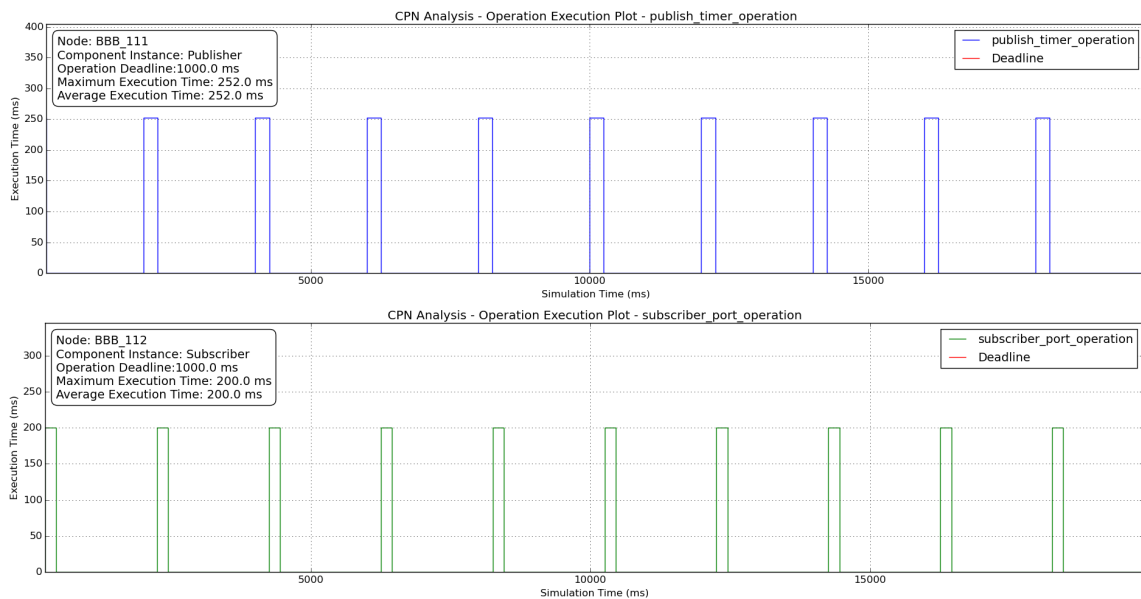

Figure 39: CPN Analysis Results: Publish-Subscribe Interactions

7.5.4.1 Bad Designs

Similar to the client-server example, we can explore another accidental bad design that may become hard to track. In a publish-subscribe interaction, the publisher and the subscriber are completely detached from each other i.e. delays in the subscriber operation do not affect the publisher. So, the only way the publisher component can affect its own behavior is how it is triggered. Periodically triggered data dissemination is the most commonly

Table 3: Publish Subscribe Example – Summary of Results

Operation Name	Component Name	Measured Response Time (μ, σ) (ms)	Measured WCRT (ms)	CPN Analysis WCRT (ms)	Deadline (ms)	Error (%)
Subscriber Port Operation	Subscriber	(194.17, 0.03)	194.71	200.0	1000	2.712
Publisher Timer Operation	Publisher	(226.48, 12.77)	249.21	252.0	1000	1.11

used streaming pattern aside from event-driven messaging. Here, if the period of the timer is decreased from 2 s to 10 ms, the publisher gets triggered too frequently and is seriously affected by a local design flaw.
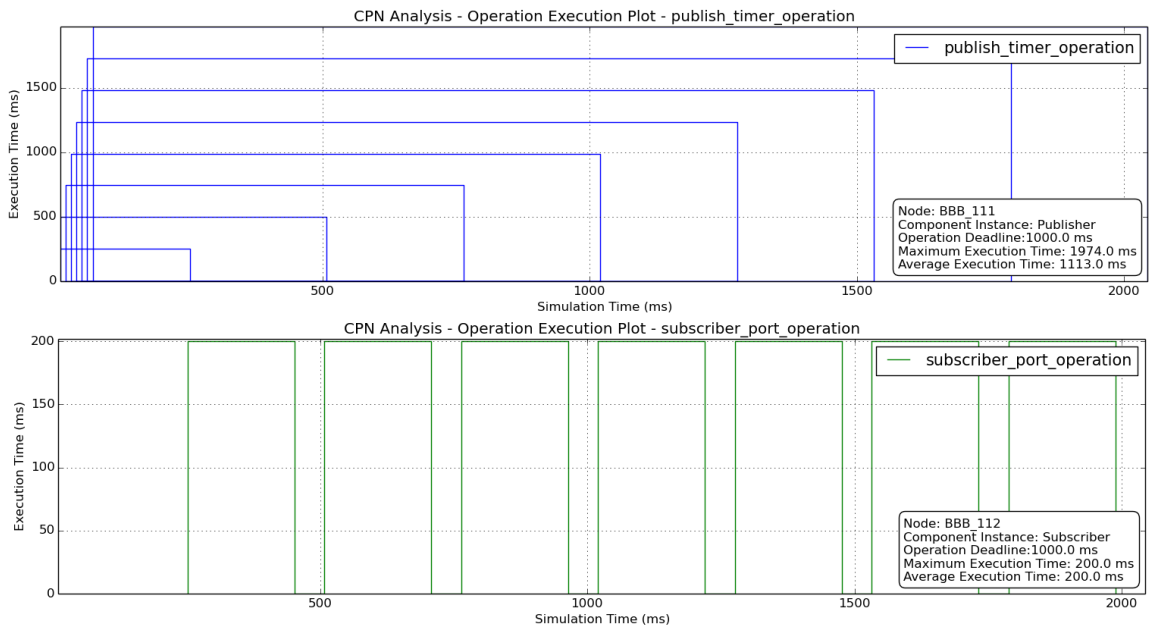

Figure 40: CPN Analysis Results: Time-triggered Publisher – Periodicity Issues

Figure 40 shows the CPN results for this design. The subscriber still performs as expected taking 200 ms to process each incoming message. But the publishing component

is affected by the periodicity of its trigger. The publish_timer fires every 10 ms, and at each expiry enqueues a timer operation request onto the publisher component's message queue. Each timer operation itself takes about 250 ms i.e. 25 times of the periodicity of its trigger. The inevitable result of this design is the monotonically rising execution times of each subsequent timer operation due to progressively delayed response. Here, the $t\_response\_time = t\_dequeue - t\_enqueue$ becomes progressively worse and eventually the publisher's message queue overflows. In the real experiment, we were unable to access the publisher component's device via remote shell as the device CPU was saturated.

7.5.5 Trajectory Planner

In the past [51], we have used a *Trajectory Planner* deployment to illustrate the utility of our state space analysis. As shown in Figure 41, a Sensor component is periodically triggered every second by the *sensor_timer* at which point it publishes a notification to the Trajectory Planner component, alerting the planner of new sensor state. The planner component receives this notification on its *state_subscriber*. On receiving this message, the planner executes a remote method invocation to the *compute* server located in the Sensor, blocked and waiting for a response. At this point, the *compute_operation* is executed on the Sensor which returns the updated sensor state. This unblocks the planner component which uses the new sensor state to perform trajectory planning tasks. This is an important scenario as it enforces all DREMS-supported interaction patterns, i.e., periodically triggered, publish-subscribe and client-server interactions. However, aside from the minor variations in the loop iteration count, there is no non-determinism in this scenario, i.e., there is deterministic order in which the various events are handled.
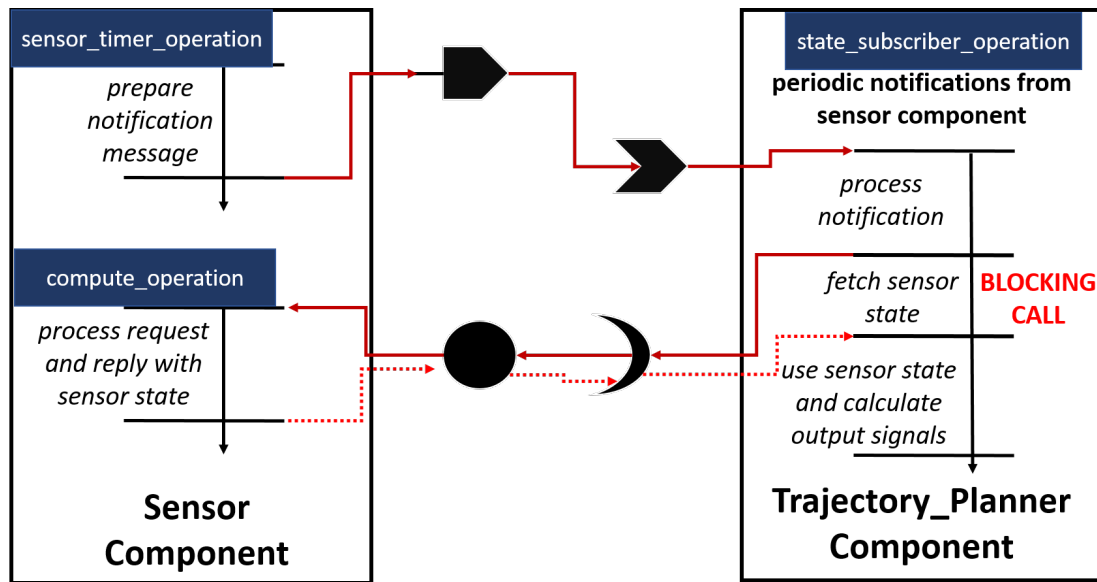
Figure 41: Trajectory Planner Test

Figure 42 shows the execution time plot of this sample where the sensor updates happen once a second. Figure 44 presents the CPN analysis plot and Table 4 summarizes the results.
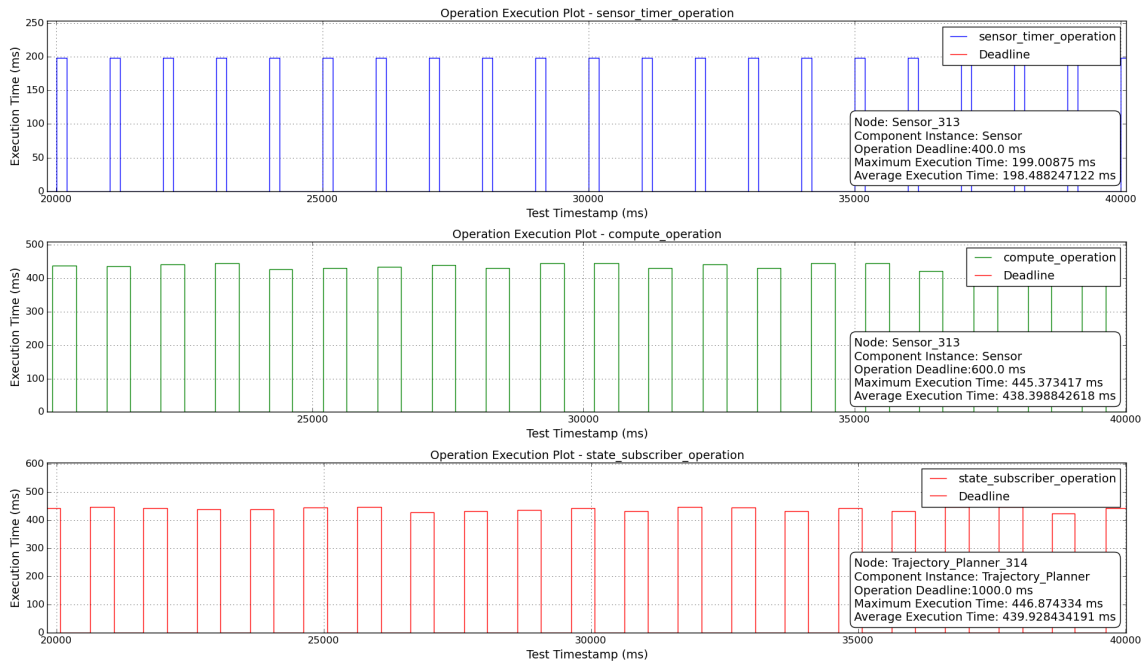
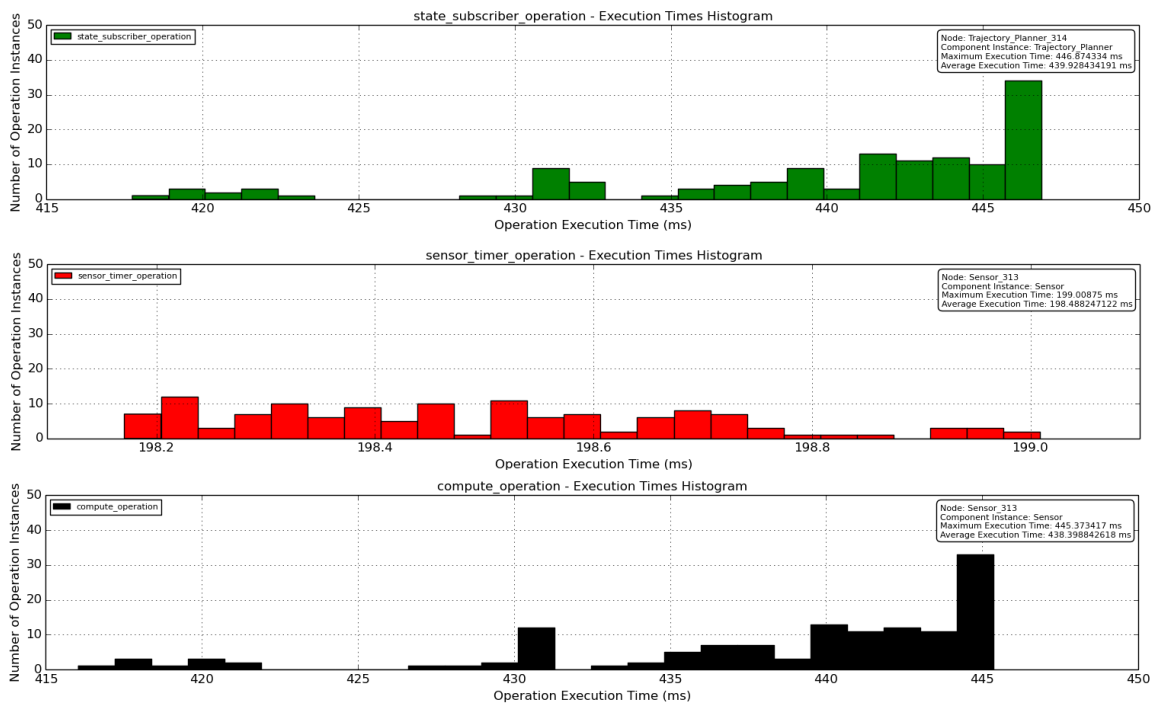**Figure 42: Experimental Observation: Trajectory Planner**

Figure 43: Histogram of Measurements Trajectory Planner

This is a common interaction pattern in Cyber-Physical systems since embedded sensors are updated at a much higher frequency than a path planning entity. Thus, the planner can query the sensor at a lower rate to sample the sensor state. In this example, the planner is matching the frequency of the sensor since the execution cost is low. However, when more components are added to this deployment, the planner would have to fetch sensor state less frequently so as to not affect other system-level deadlines.

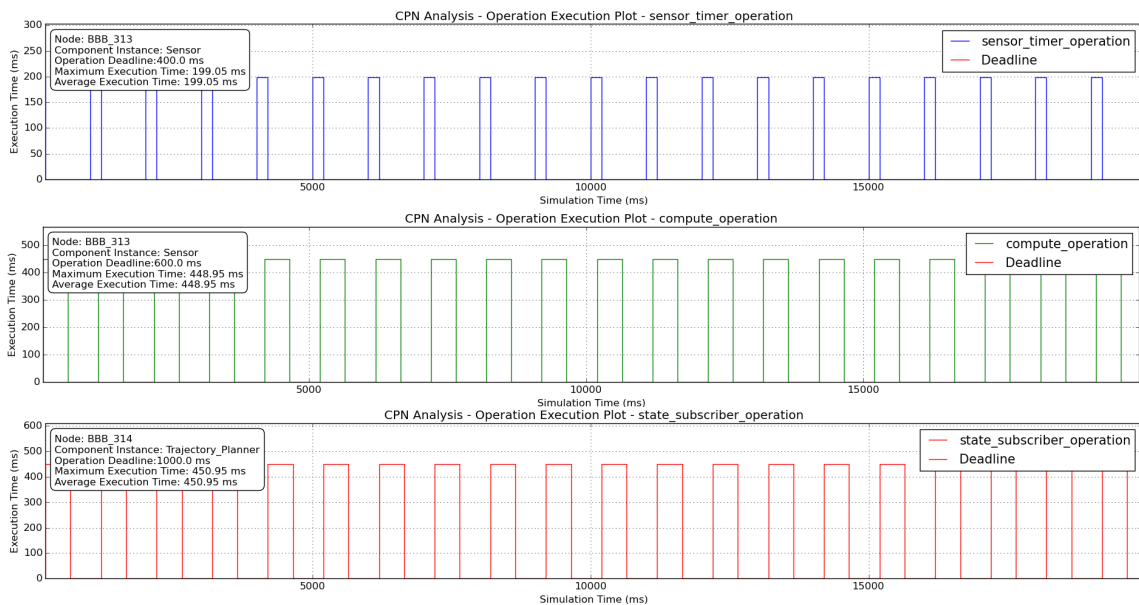**Figure 44: CPN Analysis Results: Trajectory Planner**

If the sensor update frequency is increased to 100 ms, the sensor begins to notify the planner at a much higher frequency than expected. If there is no down-sampling on the planner's side, every single update will be handled by the planner, leading to dangerous queue size growth on the planner. Figure 45 shows this deployment, as observed in the CPN analysis.

Table 4: Trajectory Planner Example – Summary of Results

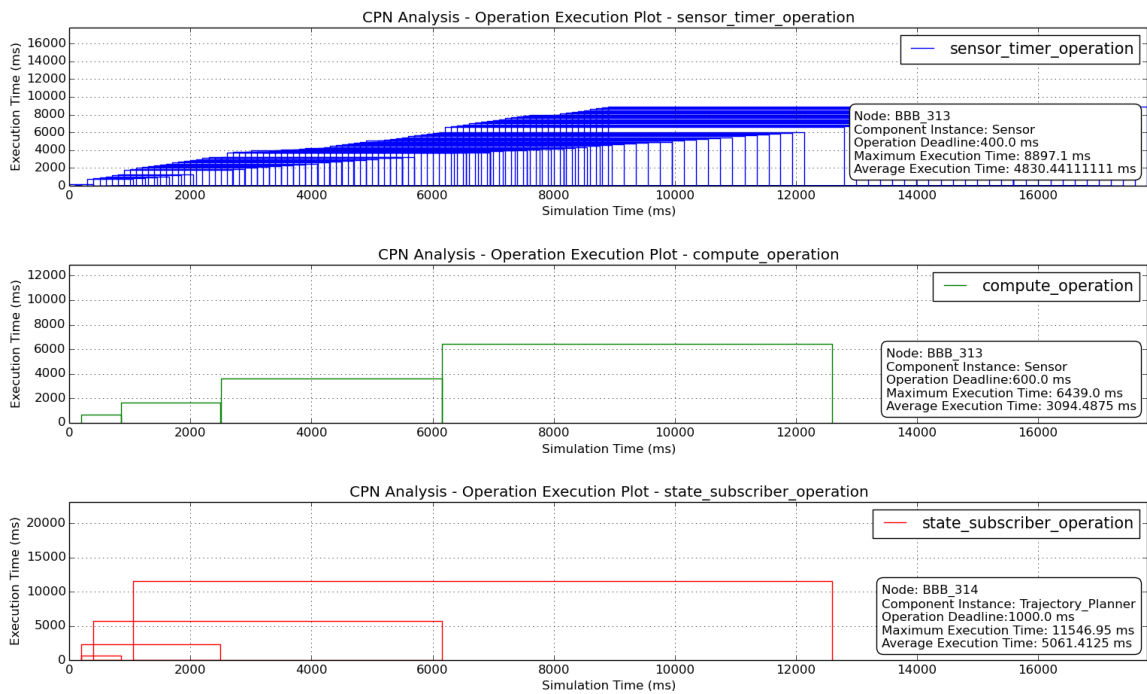| Operation Name	Component Name	Measured Response Time (μ, σ) (ms)	Measured WCRT (ms)	CPN Analysis WCRT (ms)	Deadline (ms)	Error (%)
Sensor Timer Operation	Sensor	(198.48, 0.20)	199.0	199.05	400	0.02
Compute Operation	Sensor	(438.39, 7.33)	445.37	448.95	600	0.80
State Subscriber Operation	Trajectory Planner	(439.92, 7.34)	446.87	450.95	1000	0.91

Figure 45: CPN Analysis - Sensor firing too frequently

7.5.6 Time-triggered Operations

Time-triggered operations are an integral part of our component model. DREMS components are dormant by default. A timer has to trigger a inactive component for all subsequent interactions to happen. Since the DREMS component model supports various scheduling schemes on a single component message queue, this following test evaluates a priority first-in first-out (PFIFO) scheme. Multiple timers are created in a single component, each with a unique priority and period. A timer with a high frequency is assigned a high priority. Figure 46 shows our experimental observations on a 5-timer example.
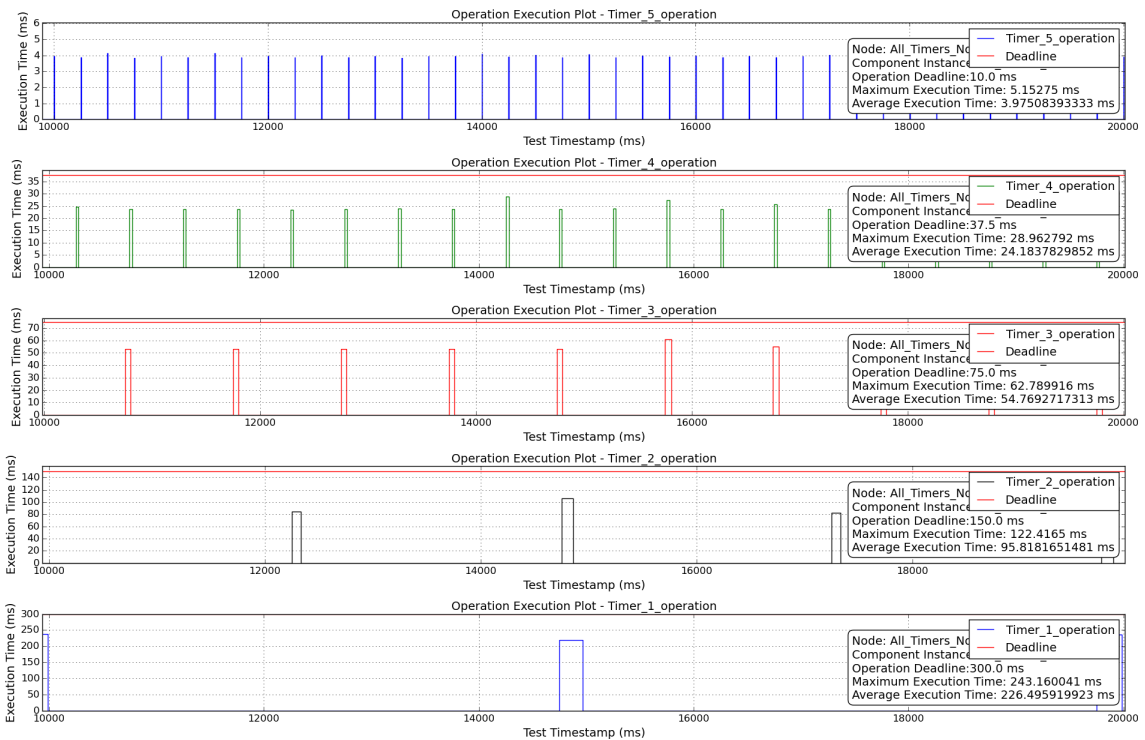
Figure 46: Experimental Observation: Periodic Timers
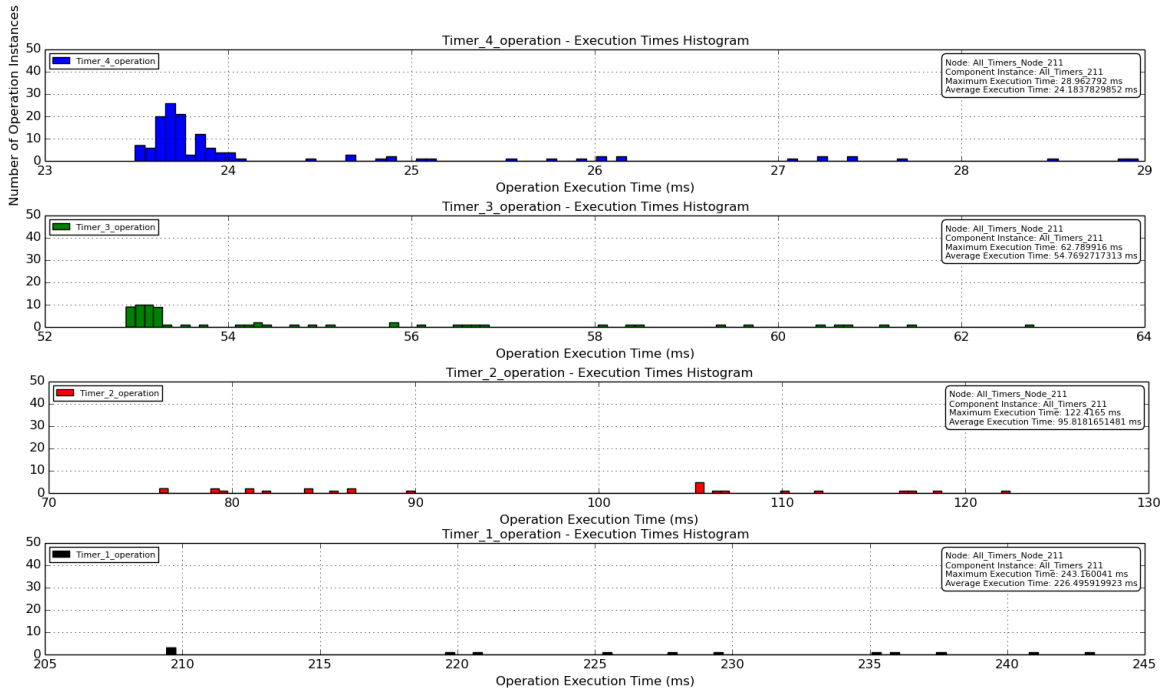
Figure 47: Histogram of Measurements for Periodic Timers Scenario

Since ROSMOD components are associated with a single executor thread and component operations are also non-preemptive, a low-priority operation could theoretically run forever, starving a higher priority operation from ever executing, leading to deadline violations e.g. *Timer_1_operation* can affect all other higher priority timers. Figure 48 shows our CPN prediction where such a scenario is evident. It can be seen that *Timer_5_operation*, the timer with the highest priority is periodically seeing spikes in execution time, courtesy of other lower priority operations consuming CPU without preemption. Table 5 summarizes the results.
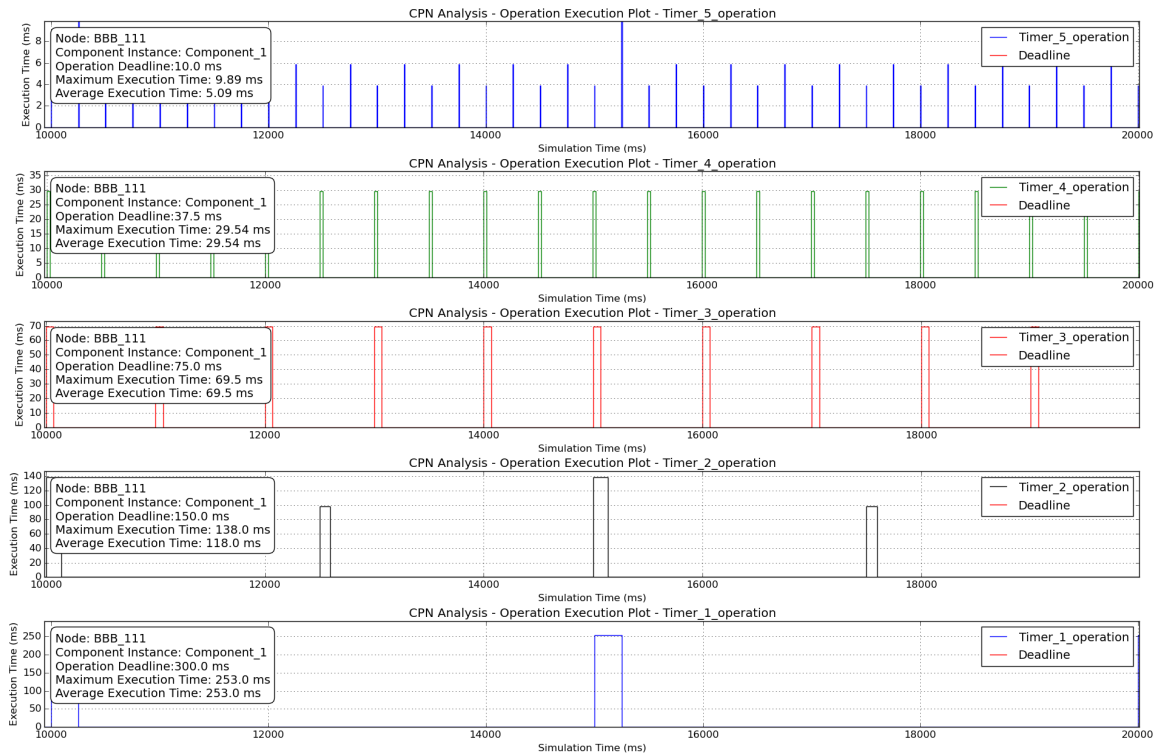
Figure 48: CPN Analysis Results: Periodic Timers

7.5.7 Long-Running Operations

Our ROSMOD component model implements a non-preemptive component operation scheduling scheme. A component operation that is in the queue, regardless of its priority, must wait for the currently executing operation to run to completion. This is a strict rule for operation scheduling and does not work best in all system designs e.g. in a long-running computation-intensive application, rejuvenating the executing operation periodically and restarting it at a previous checkpoint increases the likelihood of successfully completing the application execution. In applications executing long-running artificial intelligence (AI) search algorithms e.g. flight path planning algorithms, the computation should not hinder the prompt response requirements of highly critical operation requests such as sudden maneuver changes. Our ROSMOD component model does not support the asynchronous *cancellation* of long-running component operations to service other highly critical operations

103

Table 5: Periodic Timers – Summary of Results

Operation Name	Component Name	Measured Response Time (μ, σ) (ms)	Measured WCRT (ms)	CPN Analysis WCRT (ms)	Deadline (ms)	Error (%)
Timer_1	Component_1	(226.49, 11.53)	243.16	253.0	300	4.04
Timer_2	Component_1	(95.81, 15.01)	122.41	138.0	150	12.72
Timer_3	Component_1	(54.76, 2.64)	62.78	69.5	75	10.68
Timer_4	Component_1	(24.18, 1.13)	28.96	29.54	37.5	2.0
Timer_5	Component_1	(3.97, 0.14)	5.15	9.89	10	92.03

waiting in the queue. With a few minor modifications to our scheduling schemes, long run-ning operations can, however, be suspended if a higher priority waiting operation requires service. With these additions, we are able to model and analyze component-based systems that support long-running operations, with checkpoints, enabling the novel integration of AI-type algorithms into our design and analysis framework.

7.5.7.1 Challenges

One of the primary challenges here is to identify the semantics of a long-running com-ponent operation i.e. the scenarios under which the component operations scheduler sus-pends a cooperating long-running operation in favor of some other operation waiting in the queue. Firstly, the developer of the long-running operation must cooperate with the rest of the system for suspension i.e. at some periodic checkpoint, the operation checks to see if there are any other higher priority operations that require execution. The checkpoints have to be built into the code for long-running operations. If a long-running computation is modeled as a sequence of execution steps with bounded checkpoints, then the operation

would execute in discrete steps and suspend at such checkpoints if necessary. An important challenge here is accurately identifying the priority difference between the long-running operation and the waiting operation. If the long-running operation is one checkpoint away from completion e.g. 100-200 ms of execution time, then strictly following our suspension rules would not be the most prudent choice since this operation is almost complete. However, if the waiting operation is a critical one, then regardless of the state of the long-running operation, the executing operation must be suspended. Secondly, the modeled long-running computation semantics must be incorporated into our component model so that any analysis results obtained can be suitably validated.
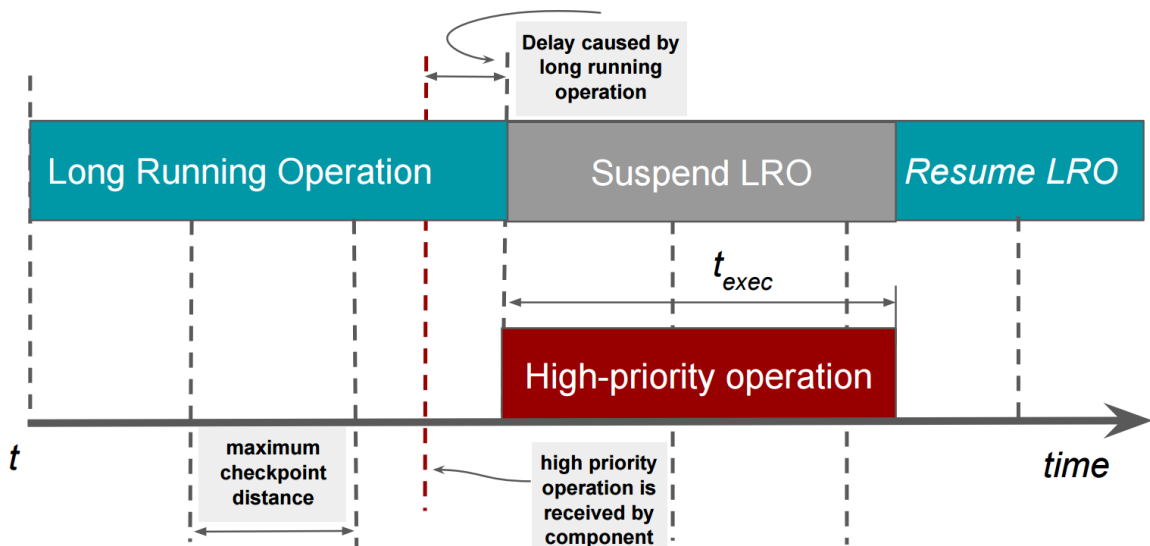
Figure 49: Long Running Operations - Timing Diagram

7.5.7.2   Implementation and Results

In each long-running operation, we, therefore, include a synchronous *checkpoint step*, as shown in Figure 49. The only assumption we make about this long-running operation is the periodicity of these checkpoint steps i.e. we know how frequently a new checkpoint is reached and we assume that the search algorithm used by the long-running operation is

capable of reaching a safe state (the checkpoint) before suspending itself if required. If a higher priority operation is ready and waiting in the queue, the long-running operation runs till the next checkpoint is reached, then suspends. The higher priority operation is then processed.

The component assembly for this test consists of three components: *Component_1*, *Component_2* and *Component_3*. Component_1 and Component_2 periodically publish messages and Component_1 has a subscriber that receives these messages at different rates. Component_1 also executes a long-running operation that yields the CPU to the higher priority subscriber operation as required. Meanwhile, Component_3 periodically queries a service provided by Component_2. Figure 50 shows the execution time plot of this scenario, as measured on our testbed. Notice the long-running operation on Component_1 consuming 64 sec of CPU during the test. This operation periodically suspends itself so that the Name_Subscriber_Operation in Component_1 can receive messages on the subscriber port.
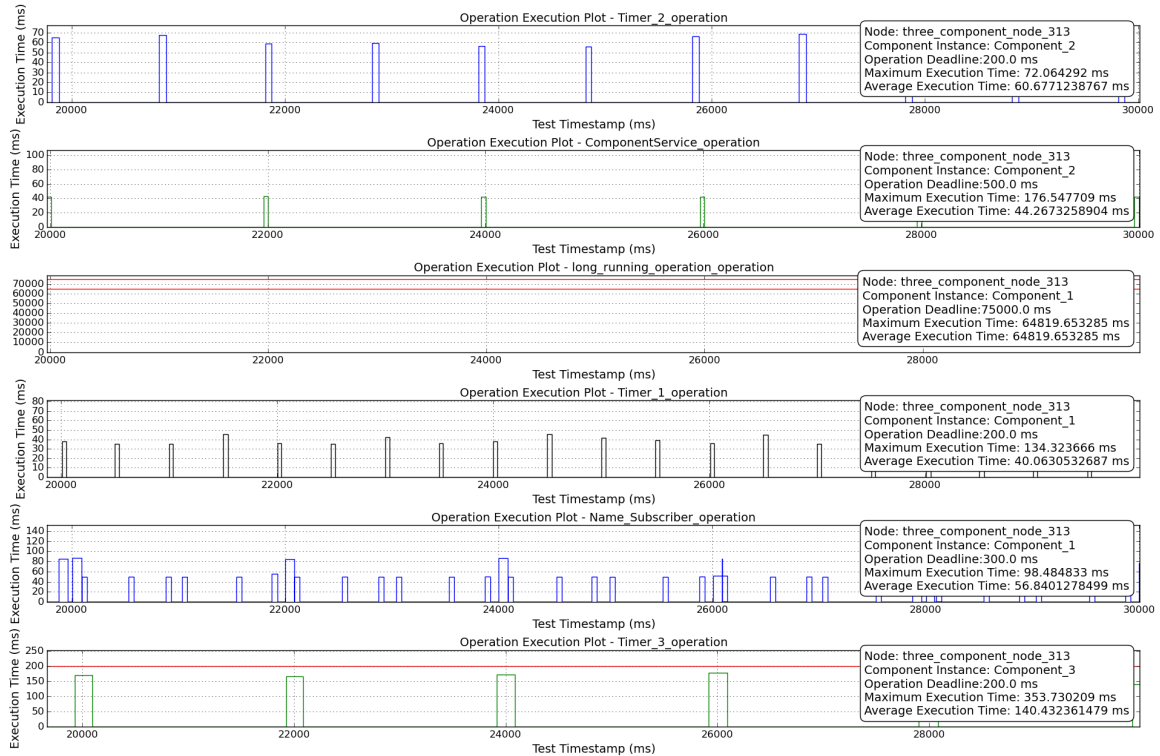
Figure 50: Experimental Observation: Composed Component Assembly.

For the CPN analysis, in order to obtain pure execution times of all these operations, each operation on each component is executed as a stand-alone function on the hardware. This way, we know the average and worst-case execution times of all operational steps with minimal interruptions. These numbers are injected into our generated CPN and state space analysis is performed. Figure 51 shows our CPN analysis results for the same assembly.
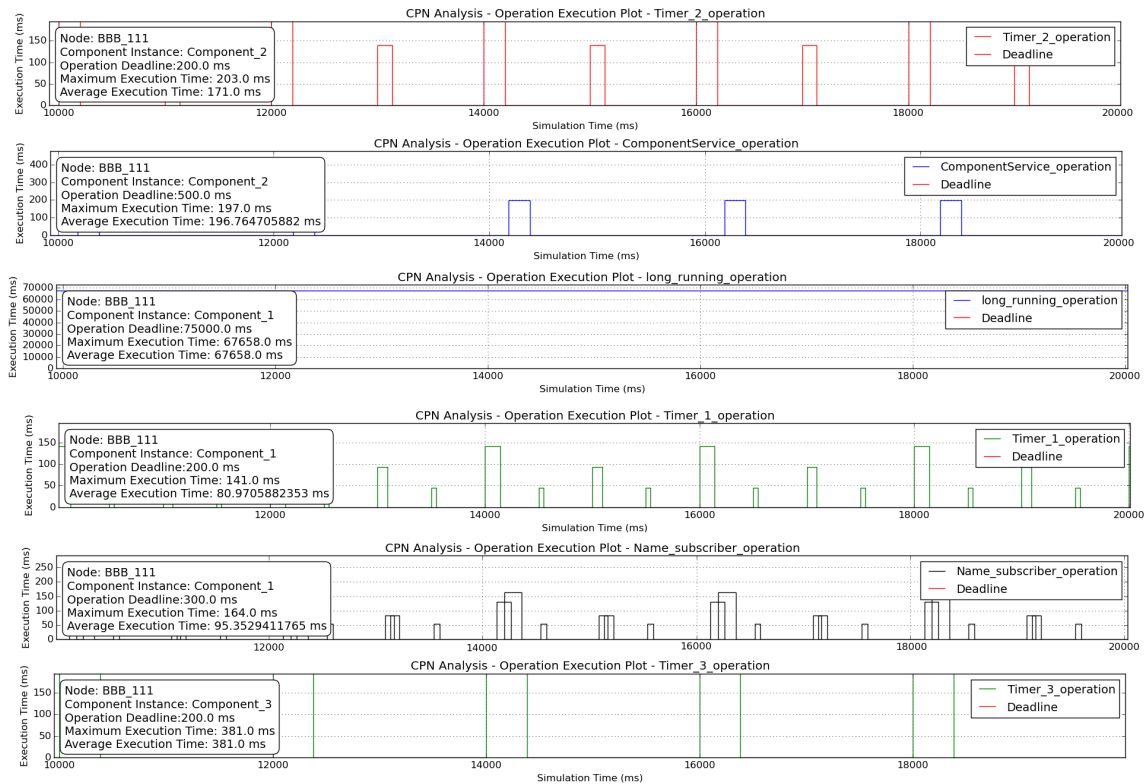
Figure 51: CPN Analysis Results: Composed Component Assembly

7.5.8 Integration with Physics Simulators - Cyber-Physical Systems Scenarios

Kerbal Space Program [3] (KSP) is a widely popular space flight simulator for a variety of platforms including Linux, OS X and Windows. In this game, players get to manage a space program, designing and building spacecrafts and exploring celestial bodies.

While KSP does not provide a perfect simulation of reality, it has been widely praised for its component-based design and development process coupled with aerodynamic, gravitational, and rigid-body interaction and simulation. In this simulation, every man-made object follows Newtonian dynamics. Rocket thrust and aerodynamic forces are accurately applied to the vehicles based on the directions and precise positions in which the force-affected elements are mounted on the vessel. Using KSP, we have modeled scenarios for a variety of flight missions including interplanetary travel. In this section, we briefly describe an aircraft flight controller that was designed and tested using the RCPS testbed and KSP.

108

This CPS scenario is a flight controller application used to completely control a KSP aircraft from the primary space-plane hanger to a destination airport. The application processes require inputs from KSP e.g. sensor data about pitch, roll, yaw, mean altitude etc. and interfaces to control the flight dynamics model by setting actuators for thrust, pitch, and heading. If these interfaces are available, then the processes can periodically retrieve flight telemetry and provide commands for course correction and feedback control.

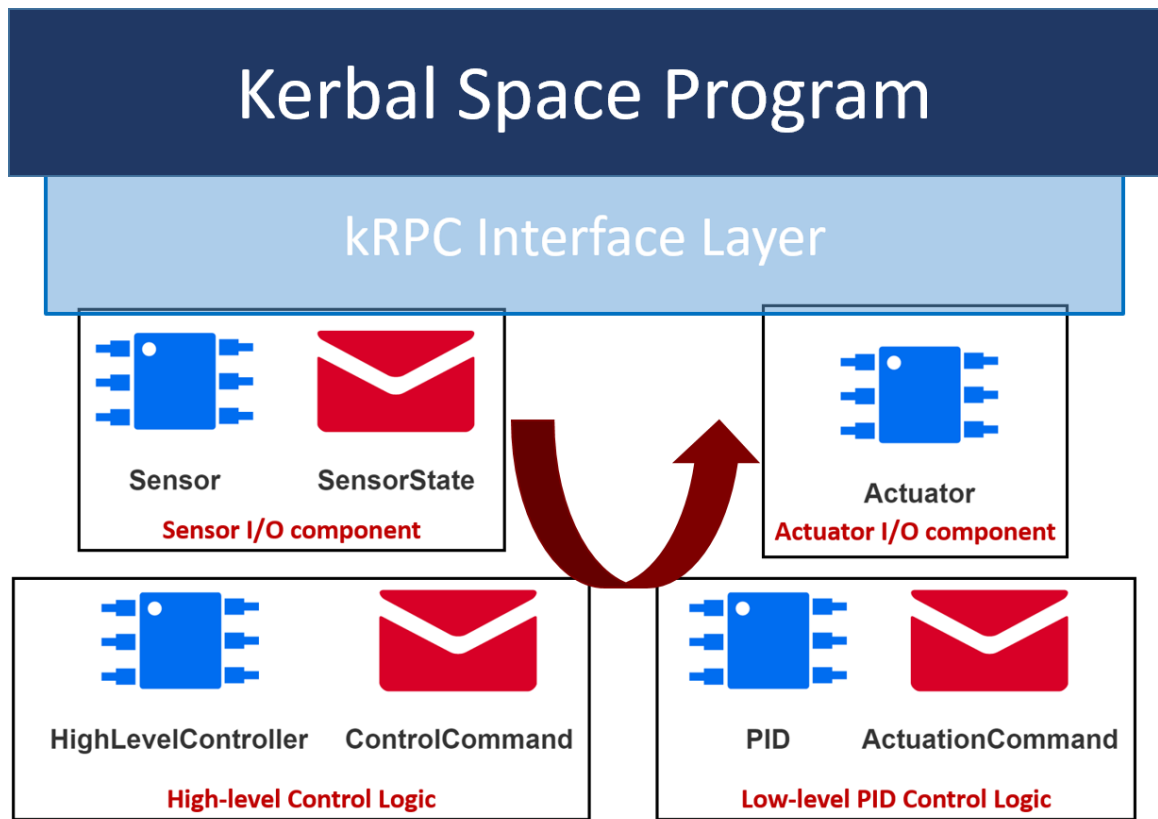**Figure 52: Kerbal Space Program - Flight Control Application - Stack**

Using an open source project called kRPC [2] (Kerbal Remote Procedure Call Server), the BBB nodes running CPS processes are provided with an interface to the simulation. All components that interact with the simulation through kRPC are characterized as *I/O components*. Figure 52 shows the software stack for this flight control application. This

test consists of four components: periodic sensor stream, a high-level controller, a low-level PID controller, and an actuator component. The sensor component interacts with the simulation and receives a stream of sensory information e.g. pitch, roll, yaw, heading, throttle, mean altitude etc. as fast as the kRPC interface can provide it. This stream of data is sampled at a frequency of 20 Hz (lower than the sensor update frequency of 60 Hz), packaged as a *Sensor_State* message and published by the Sensor component. The high-level component receives this sensor information and decides on high-level state changes e.g. take off, cruising, landing etc. Based on the required high-level state changes, this controller component commands the PID component to maneuver the flight to the right altitude and heading. The PID component uses pre-defined gains and calculates new thrust, pitch, roll and yaw values based on the next goal altitude and heading. This actuation command is published by the PID component and handled by the actuator component. The actuator component provides the second interface to KSP for the ROSMOD application by commanding the simulation interface to control the vessel. Figure 53 shows the Stearwing A300 aircraft taking off from the space-plane hanger and stabilizing at a cruising altitude of 2000 meters.
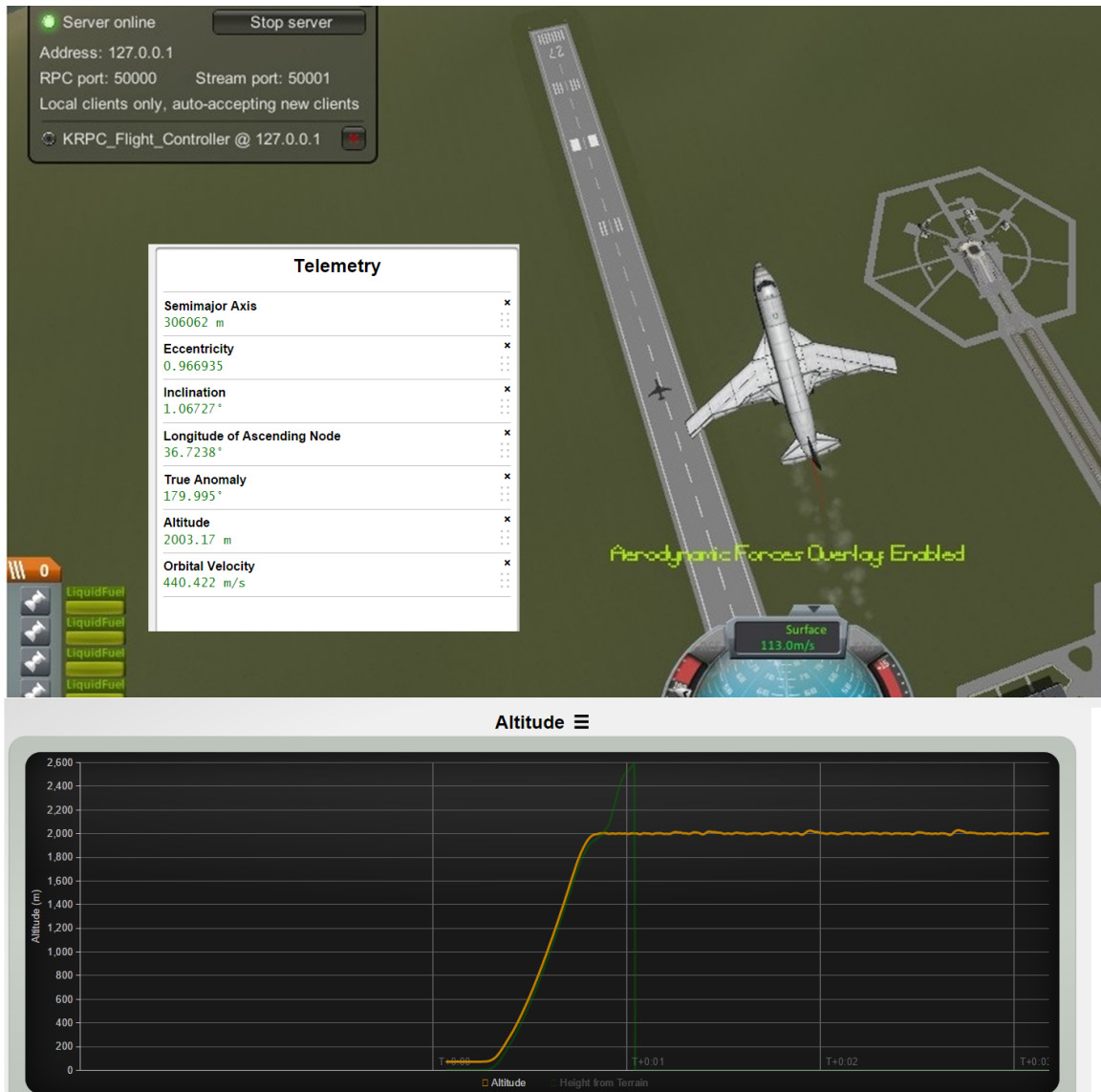
Figure 53: Stearwing A300 PID Control

Figure 54 shows about 2.5 minutes of execution of the flight control application on the experimental testbed and Figure 55 shows the CPN analysis results for the first 40 seconds of execution. Table 6 compares the response time values for all operations in this application. Many of the component operations in this sample have pure execution times in the order of hundreds of microseconds with spikes in execution times at the end of the test during teardown i.e. the I/O components do not receive prompt responses from the

simulation as the experiment is stopped. These delays propagate to other components in the application e.g. actuator control subscriber operation.
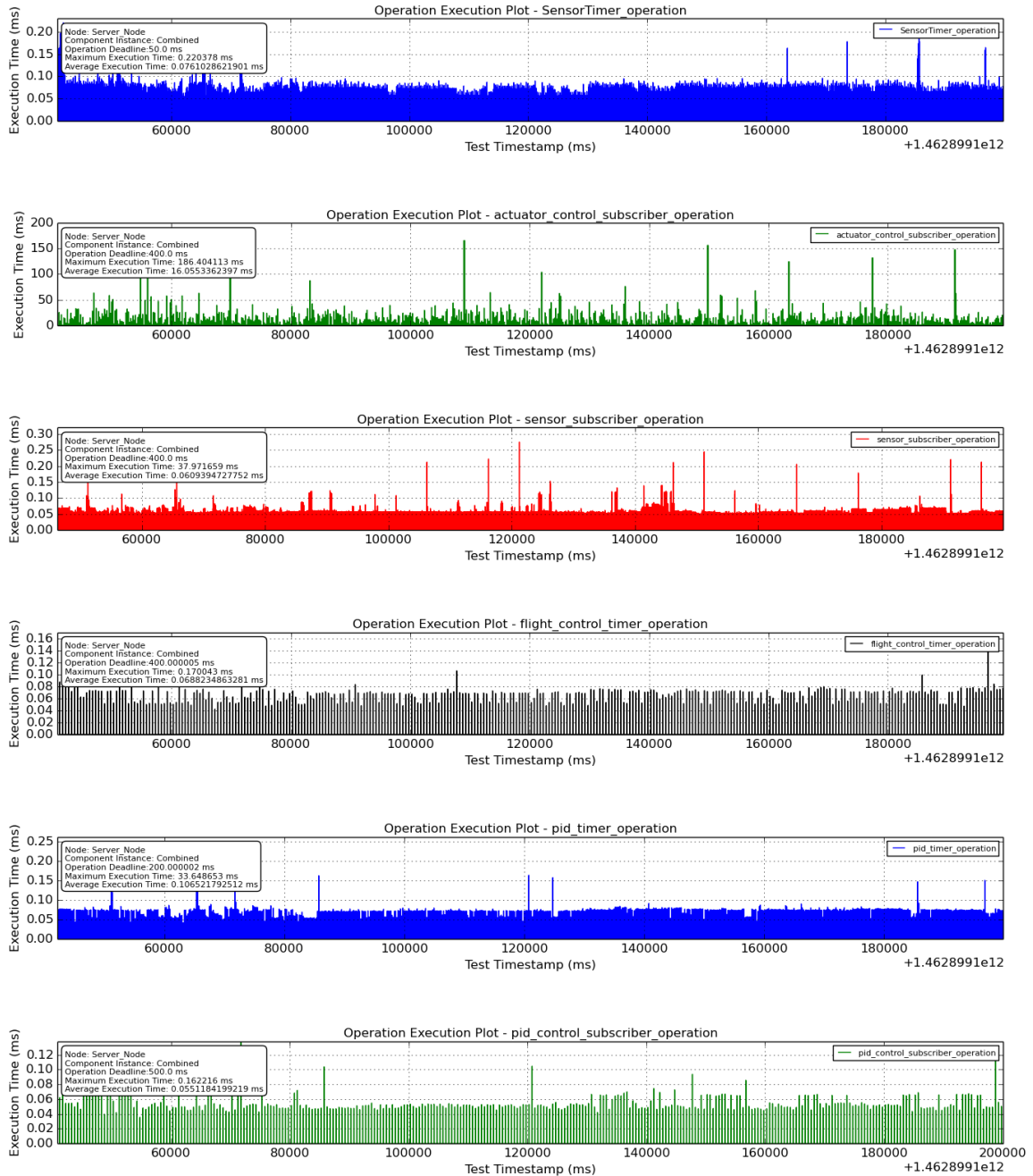
Figure 54: Stearwing Flight Control - Experimental Observations

Figure 55 shows the CPN analysis results for this application. The worst-case pure

execution times of all operation steps are calculated over multiple runs of the application and the business logic for all operations are constructed. The generated CPN is executed for a 100,000 steps i.e. about 40 seconds of thread activity. Since the period of the sensor timer is 50 ms, the analysis cover around 800 periods of sensor state changes.

Figure 55: Stearwing Control - CPN Analysis Results

Table 6: KSP Flight Controller – Summary of Results

Operation Name	Component Name	Measured Response Time (μ, σ) (ms)	Measured WCRT (ms)	CPN Analysis WCRT (ms)	Deadline (ms)	Error (%)
Sensor Timer	Sensor	(0.07, 0.016)	0.22	0.23	50	4.36
Actuator Control Subscriber	Actuator	(16.055, 17.75)	186.40	203.84	400	9.35
Sensor Subscriber	High-level Controller	(0.06, 0.37)	0.09	1.019	400	1021.01
Flight Control Timer	High-level Controller	(0.06, 0.13)	0.17	0.170	400	0.025
PID Timer	PID	(0.10, 0.96)	0.186	4.201	200	2152.5
PID Control Subscriber	PID	(0.05, 0.01)	0.162	1.013	500	524.47

As seen in the percentage error column of the above table, there are some huge over-estimates of WCRT for some component operations. The cause for this is discussed below.

7.6   Analysis Limitations

The business logic abstraction, as presented in Section 5.3, is quite simple. This model represents abstractly how long the different blocks of code in an operation take, but does not fully model its behavior e.g. each non-blocking local code block is represented by a single WCET and no other properties. Since variables local to these functional code blocks are not modeled, this language also does not model conditional behavior. So, a high-level controller component, that executes an event-driven state machine, as in the KSP test described earlier, cannot be sufficiently modeled with the current model. Moreover, depending on the local state of such code blocks, different interaction patterns may be invoked

e.g. if this high-level controller was responsible for performing periodic image processing to identify an object, the image processing (and all the associated image processing) would stop once the object was identified. Since this conditional interaction is not modeled in the CPN business-logic language, the abstraction would assume that the image processing interactions always happen, instead of conditionally. Such abstractions can lead to gross over-estimation of the worst-case execution times of component operations.

Table 6 also exposes this limitation. The *sensor subscriber operation* in the high-level controller component has an experimental response time of 90 usec, while the CPN analysis overestimates the WCRT to 1.019 msec. This subscriber operation is the one that receives sensor state updates and makes decisions regarding the next state of the system as a whole; decisions that can lead to large variabilities in end-to-end response times, as evident in Figure 54. These variabilities also propagate to other component operations and lead to over-estimation on the analysis side. This is the problem with (1) not modeling (in CPN) variables and conditional statements in the component operations and (2) always choosing the worst-case execution of steps inside a component operation. With state space analysis, if this bad execution trace can be reached, then the analysis result becomes an over-estimate.

The primary reason for keeping the business logic language to this level of simplicity is to reduce the level of state space explosion i.e. modeling all local variables of an operation, and consequently the state changes for each of these variables would exponentially increase the size of the state space to be analyzed. Also, modeling the local variable state would also require (1) modeling the semantics of language with which the operation code was written, (2) evaluating all expressions using these local variables, and (3) calculating the result of all conditionals used in the business logic at all possible times. Such an analysis would be too refined, hard to implement and susceptible to semantic errors.

CHAPTER VIII

SUMMARY AND FUTURE WORK

This thesis has presented an integrated timing analysis methodology for component-based distributed real-time embedded systems. The following list summarizes the contributions.

1. Developed a Colored Petri net-based methodology to model the temporal behavior of component-based Distributed Real-time Embedded Systems. The Colored Petri net models component-based applications using the semantics of the DREMS Component Model.

2. Developed a translation scheme that uses a precise model of the software, hardware and system deployment, to generate the Colored Petri net

3. Developed and applied state space analysis methods that answer important questions about the timing behavior of the system e.g. estimated worst-case response times, presence/absence of deadline violations, deadlocks etc.

4. Experimentally validated the design-time CPN timing analysis results by executing the component assembly in an embedded systems testbed consisting of 32 Beaglebone Black embedded boards.

There is much significance to the presented methodology. A number of existing timing analysis methods, as discussed in Chapter III, are either simulation-based analysis techniques, or state space analysis methods applied to heavily abstracted models of a large and complex system. Much of the system model is abstracted away and simple threading models, e.g., set of periodically triggered threads with fixed WCET, are considered. This level of abstraction is often necessary to keep the analysis tractable, but this is also

due to the modeling paradigm being used for analysis. By using high-level Petri nets like CPN, a considerable amount of information can be modeled concisely in a scalable manner. More importantly, the problem with many of these existing methods is the abstraction itself. For a complex, hierarchically controlled system such as DREMS where there are various factors of influence, e.g., (1) OS temporal partitioning, (2) component-level operation scheduling, (3) distributed interacting component assemblies, etc., there really aren't any tools that address the problem of design-time timing analyis and verification with the necessary levels of detail. Unlike the simple threading models and abstractions typically considered for analysis, the DREMS component threads cannot be associated with a single execution time. In fact, the thread behaviors are almost entirely dependent on the operation scheduling, which operates at a much higher level of abstraction. Being able to account for hierarchical scheduling, varied interaction patterns, temporal partitioning, preemptive and non-preemptive scheduling, distributed deployments and configurations, scheduling schemes such as FIFO, PFIFO and EDF on a component-level, and provide scalable and well integrated timing analysis is the primary significance of this work. A modern industrial system is going to present design challenges in any if not all of the above aspects. The ability to generate a structured, hierarchical analysis model from an annotated system model is also an advantage here. Unlike other Petri net analysis models, our analysis model remains structurally the same; the properties of the system model are encoded as CPN tokens with the execution semantics implemented in Standard ML as CPN functions and the equal priorities of CPN transitions covering the possible non-deterministic behaviors.

There are still some potential extensions to this work. All of the results presented in this thesis make an important assumption about the network - the network resources available to each component are much larger than the requirements of the application i.e. there are no buffering delays on the network queues when components periodically produce data. The current analysis model, in this respect, is quite lacking. When a component publishes a message on a topic, the analysis immediately generates a reception message that waits to

enqueue on the receiver's message queue. In reality, this interaction could be a lot more involved - the published message is sent to the kernel network queue on the sender's side and removed from this queue following a data production profile i.e. available bandwidth as a function of time. When dequeued, the packets take a finite worst-case transmission time before being noticed on the receiver's side. The buffering delays on the sender's side and the transmission time on the network are completely ignored by the timing analysis model. In order to improve on this design, we have attempted to integrate existing Network Calculus-based analysis methods [28] into our CPN. Specifically, a place is added to model the Network Queue and a Dequeue transition fires when the network is ready to transport more packets from the sender. The dequeuing follows a strict network profile and ceases transmission when the data production rate is larger than the available bandwidth.

Also, the business logic model i.e. the model of execution code in a component operation is quite simplistic. This model is able to represent local non-blocking code blocks, DREMS-style interaction patterns, and bounded loops of either. The model is however, unable to represent conditional statements that rely on local variables. This is a disadvantage as the resultant analysis is unable to accurately represent the execution behavior of the runtime code. Many if not most distributed real-time embedded scenarios in real-life exhibit conditional behavior that are driven by runtime state e.g. robotic applications that transition from one state of operation to another at runtime. The business logic of such operations cannot be fully represented by the current model and this has lead to gross overestimates in execution time behavior. Such over-estimation makes the analysis results useless as no operation scheduling may be calculated as feasible. Thus, the business logic model, and its integration into the analysis model both require improvement in order to support a wider range of execution scenarios.

Appendices

APPENDIX A

PUBLICATIONS

The full text in each of the following papers was reviewed by at least 3 reviewers.

1.1 Workshop Papers

- P. S. Kumar, A. Dubey, and G. Karsai. Colored petri net-based modeling and formal analysis of component-based applications. In *11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2014*, page 79, 2014

- P. Kumar and G. Karsai. Integrated analysis of temporal behavior of component-based distributed real-time embedded systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2015 IEEE International Symposium on Real-time Computing (ISORC)*, pages 50–57, April 2015

- P. Kumar and G. Karsai. Experimental validation of timing analysis for component-based distributed real-time embedded systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2016 IEEE International Symposium on Real-time Computing (ISORC)*, May 2016

1.2 Conference Papers

- P. Kumar, W. Emfinger, A. Kulkarni, G. Karsai, D. Watkins, B. Gasser, C. Ridgewell, and A. Anilkumar. ROSMOD: A Toolsuite for Modeling, Generating, Deploying, and Managing Distributed Real-time Component-based Software using ROS. In *Proceedings of the IEEE Rapid System Prototyping*, RSP 2015, Amsterdam, Netherlands, 2015. IEEE

- P. Kumar, W. Emfinger, and G. Karsai. A Testbed to Simulate and Analyze Resilient

Cyber-Physical Systems. In *Proceedings of the IEEE Rapid System Prototyping*, RSP 2015, Amsterdam, Netherlands, 2015. IEEE

- W. Emfinger, P. Kumar, A. Dubey, W. Otte, A. Gokhale, G. Karsai. DREMS: A Toolchain for the Rapid Application Development, Integration, and Deployment of Managed Distributed Real-time Embedded Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS@Work 2013, Vancouver, Canada, 2013. IEEE

- Balasubramanian, D., W. Emfinger, P. S. Kumar, W. Otte, A. Dubey, and G. Karsai. An application development and deployment platform for satellite clusters. In *Proceedings of the Workshop on Spacecraft Flight Software*, 2013

- Balasubramanian, D., A. Dubey, W. R. Otte, W. Emfinger, P. Kumar, and G. Karsai. A Rapid Testing Framework for a Mobile Cloud Infrastructure. In *Proceedings of the IEEE International Symposium on Rapid System Prototyping*, RSP, 2014. IEEE

1.3   Journal Papers

- P. Kumar, W. Emfinger, G. Karsai, D. Watkins, B. Gasser, and A. Anilkumar. ROS-MOD: A Toolsuite for Modeling, Generating, Deploying, and Managing Distributed Real-time Component-based Software using ROS. In special issue of *Journal of Electronics* on Rapid System Design with Dedicated Architectures and Specific Software Tools, 2016.

- D. Balasubramanian, A. Dubey, W. Otte, T. Levendovszky, A. Gokhale, P. Kumar, W. Emfinger, and G. Karsai. Drems ml: A wide spectrum architecture design language for distributed computing platforms. *Science of Computer Programming*, 2015

- Levendovszky, T., A. Dubey, W. R. Otte, D. Balasubramanian, A. Coglio, S. Nyako,

W. Emfinger, P. Kumar, A. Gokhale, and G. Karsai. DREMS: A Model-Driven Distributed Secure Information Architecture Platform for Managed Embedded Systems. In *IEEE Software*, vol. 99: IEEE Computer Society, 2014. IEEE

REFERENCES

[1] Beaglebone Black. http://beagleboard.org/BLACK/.

[2] Kerbal Remote Procedure Call Server. https://github.com/djungelorm/krpc/.

[3] Kerbal Space Program. https://kerbalspaceprogram.com/en/.

[4] NASA CubeSat Launch initiative. https://www.nasa.gov/directorates/heo/home/CubeSats_initiative.html.

[5] NASA CubeSats Mission to Mars. http://www.jpl.nasa.gov/cubesat/missions/marco.php.

[6] B. Alpern and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Trans. Program. Lang. Syst.*, 11(1):147–167, Jan. 1989.

[7] ARINC Incorporated, Annapolis, Maryland, USA. *Document No. 653: Avionics Application Software Standard Inteface (Draft 15)*, Jan. 1997.

[8] K. J. Åström and T. Hägglund. *Advanced PID control*. ISA-The Instrumentation, Systems, and Automation Society; Research Triangle Park, NC 27709, 2006.

[9] Autosar GbR. AUTomotive Open System ARchitecture. http://www.autosar.org/.

[10] D. Balasubramanian, A. Dubey, W. Otte, T. Levendovszky, A. Gokhale, P. Kumar, W. Emfinger, and G. Karsai. Drems ml: A wide spectrum architecture design language for distributed computing platforms. *Science of Computer Programming*, 2015.

[11] F. Bause and P. S. Kritzinger. *Stochastic Petri Nets*. Springer, 1996.

[12] D. Bell. Uml basics: An introduction to the unified modeling language. 2003.

[13] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.

[14] S. Beydeda, M. Book, V. Gruhn, et al. *Model-driven software development*, volume 15. Springer, 2005.

[15] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Foundations of empirical software engineering: the legacy of Victor R. Basili*, 426:37, 2005.

[16] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[17] M. Broy and K. Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement.* Springer Science & Business Media, 2012.

[18] M. Burke and N. Audsley. Distributed fault-tolerant avionic systems-a real-time perspective. *arXiv preprint arXiv:1004.1324*, 2010.

[19] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *2009 IEEE 31st International Conference on Software Engineering*, pages 463–473. IEEE, 2009.

[20] S. Clemens, G. Dominik, and M. Stephan. Component software: beyond object-oriented programming, 1998.

[21] K. Correll, N. Barendt, and M. Branicky. Design considerations for software only implementations of the ieee 1588 precision time protocol. In *Conference on IEEE*, volume 1588, pages 11–15, 2005.

[22] M. A. Cusumano. Reflections on the toyota debacle. *Commun. ACM*, 54(1):33–35, Jan. 2011.

[23] R. David and H. Alla. Petri nets for modeling of dynamic systems: A survey. *Automatica*, 30(2):175–202, 1994.

[24] G. A. A. F. De Cindio and G. Rozenberg. Object-oriented programming and petri nets. 2001.

[25] A. Dubey, W. Emfinger, A. Gokhale, G. Karsai, W. Otte, J. Parsons, C. Szabo, A. Coglio, E. Smith, and P. Bose. A Software Platform for Fractionated Spacecraft. In *Proceedings of the IEEE Aerospace Conference, 2012*, pages 1–20, Big Sky, MT, USA, Mar. 2012. IEEE.

[26] A. Dubey, A. Gokhale, G. Karsai, W. Otte, and J. Willemsen. A Model-Driven Software Component Framework for Fractionated Spacecraft. In *Proceedings of the 5th International Conference on Spacecraft Formation Flying Missions and Technologies (SFFMT)*, Munich, Germany, May 2013. IEEE.

[27] A. Dubey, G. Karsai, and N. Mahadevan. A Component Model for Hard Real-time Systems: CCM with ARINC-653. *Software: Practice and Experience*, 41(12):1517–1550, 2011.

[28] W. Emfinger, G. Karsai, A. Dubey, and A. Gokhale. Analysis, verification, and management toolsuite for cyber-physical applications on time-varying networks. In *Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, CyPhy '14, pages 44–47, New York,

NY, USA, 2014. ACM.

[29] T. L. et al. Distributed real-time managed systems: A model-driven distributed secure information architecture platform for managed embedded systems. *IEEE Software*, 31(2):62–69, 2014.

[30] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[31] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical Report ADA455842, DTIC Document, 2006.

[32] M. Feilkas, A. Fleischmann, C. Pfaller, M. Spichkova, D. Trachtenherz, et al. A top-down methodology for the development of automotive software. 2009.

[33] D. M. Gabbay, I. Hodkinson, M. Reynolds, and M. Finger. *Temporal logic: mathematical foundations and computational aspects*, volume 1. Clarendon Press Oxford, 1994.

[34] C. Girault and R. Valk. *Petri nets for systems engineering: a guide to modeling, verification, and applications*. Springer Science & Business Media, 2013.

[35] M. Gonzalez Harbour, J. Gutierrez Garcia, J. Palencia Gutierrez, and J. Drake Moyano. Mast: Modeling and analysis suite for real time applications. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 125–134, 2001.

[36] J. Hayman and G. Winskel. Symmetry in petri nets.

[37] G. T. Heineman and W. T. Councill. Component-based software engineering. *Putting the Pieces Together, Addison-Westley*, 2001.

[38] L. E. Holloway, B. H. Krogh, and A. Giua. A survey of petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems*, 7(2):151–190, 1997.

[39] F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic lights-an autofocus case study. In *csd*, page 282. IEEE, 1998.

[40] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutofocusâĂŤa tool for distributed systems specification. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 467–470. Springer, 1996.

[41] K. Jensen. Condensed state spaces for symmetrical coloured petri nets. *Formal Methods in System Design*, 9(1-2):7–40, 1996.

[42] K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modelling and Validation of*

Concurrent Systems. Springer, 2009.

[43] K. Jensen and G. Rozenberg. *High-level Petri nets: theory and application*. Springer Science & Business Media, 2012.

[44] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[45] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A practitionerâĂŹs handbook for real-time analysis: guide to rate monotonic analysis for real-time systems*. Springer Science & Business Media, 2012.

[46] L. M. Kristensen. State space methods for coloured petri nets. *DAIMI Report Series*, 29(546), 2000.

[47] P. Kumar, W. Emfinger, and G. Karsai. Testbed to simulate and analyze resilient cyber-physical systems. In *Rapid System Prototyping, 2015. RSP '15.*, October 2015.

[48] P. Kumar, W. Emfinger, A. Kulkarni, G. Karsai, D. Watkins, B. Gasser, C. Ridgewell, and A. Anilkumar. Rosmod: A toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ros. In *Rapid System Prototyping, 2015. RSP '15.*, October 2015.

[49] P. Kumar and G. Karsai. Integrated analysis of temporal behavior of component-based distributed real-time embedded systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2015 IEEE International Symposium on Real-time Computing (ISORC)*, pages 50–57, April 2015.

[50] P. Kumar and G. Karsai. Experimental validation of timing analysis for component-based distributed real-time embedded systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2016 IEEE International Symposium on Real-time Computing (ISORC)*, May 2016.

[51] P. S. Kumar, A. Dubey, and G. Karsai. Colored petri net-based modeling and formal analysis of component-based applications. In *11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2014*, page 79, 2014.

[52] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, 2001.

[53] F. J. Lin, P. Chu, and M. T. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. In *ACM SIGCOMM Computer Communication Review*, volume 17, pages 126–135. ACM, 1987.

[54] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[55] F. Liu, A. Narayanan, and Q. Bai. Real-time systems. 2000.

[56] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Leven-dovszky, and Á. Lédeczi. Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure. In *MPM@ MoDELS*, pages 41–60, 2014.

[57] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets*. John Wiley & Sons, Inc., 1994.

[58] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[59] K. L. McMillan. *Symbolic model checking*. Springer, 1993.

[60] K. L. McMillan. Getting started with smv. *Cadence Berkeley Laboratories*, 1999.

[61] J. L. Medina and A. G. Cuesta. From composable design models to schedulability analysis with uml and the uml profile for marte. *SIGBED Rev.*, 8(1):64–68, Mar. 2011.

[62] R. Milner. *The definition of standard ML: revised*. MIT press, 1997.

[63] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[64] N. Navet and F. Simonot-Lion. *Automotive embedded systems handbook*. CRC press, 2008.

[65] Object Management Group. *UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)*, OMG Document realtime/05-02-06 edition, May 2005.

[66] W. R. Otte, A. Dubey, S. Pradhan, P. Patil, A. Gokhale, G. Karsai, and J. Willemsen. F6COM: A Component Model for Resource-Constrained and Dynamic Space-Based Computing Environment. In *Proceedings of the 16th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC '13)*, Paderborn, Germany, June 2013.

[67] J. C. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 26–37. IEEE, 1998.

[68] J. C. Palencia and M. G. Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 328–339. IEEE, 1999.

[69] D. Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification*, pages 409–423. Springer, 1993.

[70] J. L. Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977.

[71] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[72] A. Rae, P. Robert, and H.-L. Hausen. *Software Evaluation for Certification*. McGraw-Hill, Inc., 1994.

[73] R. R. Raje, J. I. Williams, and M. Boyles. Asynchronous remote method invocation (armi) mechanism for java. *Concurrency - Practice and Experience*, 9(11):1207–1211, 1997.

[74] S. R. Rakitin. *Software verification and validation for practitioners and managers*. Artech House, Inc., 2001.

[75] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*, ICATPN'03, pages 450–462, Berlin, Heidelberg, 2003. Springer-Verlag.

[76] W. Reisig. *Petri nets: an introduction*, volume 4. Springer Science & Business Media, 2012.

[77] X. Renault, F. Kordon, and J. Hugues. Adapting models to model checkers, a case study : Analysing AADL using time or colored petri nets. In *Rapid System Prototyping, 2009. RSP '09. IEEE/IFIP International Symposium on*, pages 26–33, June 2009.

[78] X. Renault, F. Kordon, and J. Hugues. From aadl architectural models to petri nets: Checking model viability. In *Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC '09. IEEE International Symposium on*, pages 313–320, March 2009.

[79] V. Santiago, A. S. M. Do Amaral, N. L. Vijaykumar, M. d. F. Mattiello-Francisco, E. Martins, and O. C. Lopes. A practical approach for automated test case generation using statecharts. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 2, pages 183–188. IEEE, 2006.

[80] B. Selic. A generic framework for modeling resources with uml. *Computer*, 33(6):64–69, 2000.

[81] D. C. Sharp. Reducing avionics software cost through component based product

line development. In *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*, volume 2, pages G32–1. IEEE, 1998.

[82] M. Simulink and M. Natick. The mathworks, 1993.

[83] A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(4):702–734, 2004.

[84] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and microprogramming*, 40(2):117–134, 1994.

[85] A. Valmari. A stubborn attack on state explosion. In *Computer-Aided Verification*, pages 156–165. Springer, 1990.

[86] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[87] J. M. Voas and A. K. Ghosh. System and method for software certification, Mar. 1 2005. US Patent 6,862,696.

[88] J. Waldo. Remote procedure calls and java remote method invocation. *Concurrency, IEEE*, 6(3):5–7, 1998.

[89] J. Wang. *Timed Petri nets: Theory and application*, volume 9. Springer Science & Business Media, 2012.

[90] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill. QoS-enabled Middleware. In Q. Mahmoud, editor, *Middleware for Communications*, pages 131–162. Wiley and Sons, New York, 2004.

[91] M. Westergaard, S. Evangelista, and L. M. Kristensen. Asap: an extensible platform for state space analysis. In *Applications and Theory of Petri Nets*, pages 303–312. Springer, 2009.

[92] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problemâĂŤoverview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[93] G. Winskel. Events, causality and symmetry. *The Computer Journal*, page bxp052, 2009.

[94] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *CONCUR'93*, pages 233–246. Springer, 1993.

[95] T. Ylonen and C. Lonvick. The secure shell (ssh) protocol architecture. 2006.

[96] M. Zhou and K. Venkatesh. *Modeling, simulation, and control of flexible manufacturing systems: a Petri net approach*, volume 6. World Scientific, 1999.

[97] A. Zimmermann and G. Hommel. A train control system case study in model-based real time system design. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8–pp. IEEE, 2003.

[98] A. Zoitl. *Real-time Execution for IEC 61499*. ISA, 2008.

[99] W. Zuberek. Timed petri nets definitions, properties, and applications. *Microelectronics Reliability*, 31(4):627–644, 1991.

[100] R. Zurawski and M. Zhou. Petri nets and industrial applications: A tutorial. *Industrial Electronics, IEEE Transactions on*, 41(6):567–583, 1994.