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

AUGUST, 2016

Nashville, Tennessee

Approved: Date:

_____ _____

Dr. Gabor Karsai

_____ _____

Dr. Xenofon D. Koutsoukos

_____ _____

Dr. Gautam Biswas

_____ _____

Dr. Akos Ledeczi

_____ _____

Dr. Bharat Bhuva

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 (DRE) systems is becoming increasingly recognized. Embedded software is pertinent in a variety of heterogeneous domains e.g. avionics [19], locomotive [119], and industrial control systems [120]. The volume and complexity of such software grows everyday depending on an assortment of factors, including challenging system requirements e.g. resilience to hardware and software faults, remote deployment and repair. The process of large scale deployment of embedded software, for this reason, has become considerably more arduous - periodic peer reviews, numerous verification and certification methods 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 [25]. In 2010, Toyota recalled some 10 million vehicles, an extraordinary number given that the company sold only about seven million vehicles during that period. Toyota engineers described the problem as a disconnect in the vehicle's complex anti-lock brake system (ABS) that causes 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 tens of 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) and development [15, 24, 42] 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 class models, 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 them composed together to derive a more robust system.

Remote embedded devices e.g. fractionated spacecraft following mission timelines and hosting distributed software applications expose several concerns including strict timing requirements, complexity in deployment, repair and integration; and resilience to 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 ***D**istributed **RE**al-time **M**anaged **S**ystem* (DREMS) [29, 32] 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 [30], and a component model [83] 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* [17] 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 of each stage is hidden, the large majority of industrial software development follows this lifecycle. Embedded software development, especially for safety critical systems, does not lend itself well to this lifecycle. The analysis presented in this work, supports and argues for a verification-driven workflow, as shown in Figure 1b. Application developers use domain-specific modeling languages to structure large-scale component assemblies and modular code generation features to speed up software development efforts. Moreover, domain-specific properties such as interaction patterns, component execution code, and

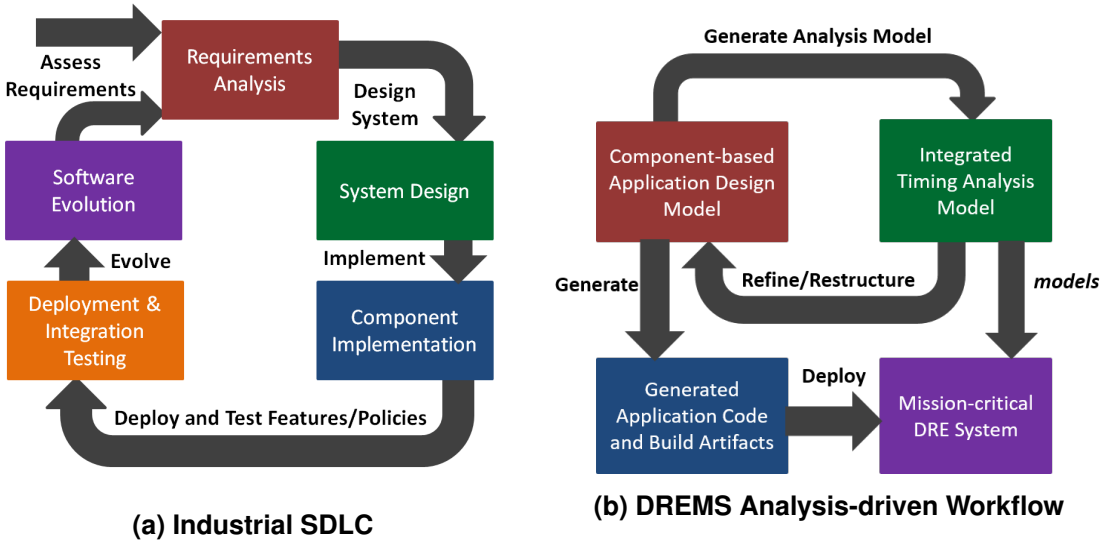

Figure 1: Embedded Software Development Lifecycle Comparison

associated temporal properties such as worst-case execution times, deadlines etc. can be easily injected into such models. Using such application parameters in the *design* model, a Colored Petri net-based (CPN) [51] *formal analysis model* is generated. The system behavior is both simulated and analyzed using a CPN execution engine, CPN Tools [95], and useful properties of the system are verified. By generating a bounded *state space* of the system, the execution traces exhibited by the system are 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 structure of 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 tightly 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. Chapter **??** concludes the dissertation, providing a summary of the detailed work. Finally, Chapter **??** lists relevant publications.

CHAPTER II

FUNDAMENTALS

A real-time system [69] 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 in 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 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 digital 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. 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 assumed 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 do not degrade the overall system performance e.g. delays in video streaming services. Hard real-time systems, however, are systems where missed deadlines could have critical, potentially fatal consequences e.g. response times on brake pedals in vehicles. Thus, hard real-time systems are by definition safety critical. 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. The required

error-detection latency must 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.

A real-time system can be decomposed into three communicating entities; (1) a controlled object; could be a physical subsystem, (2) a (potentially distributed) computer system executing programs, and (3) a human user. The computer system consists of computational nodes that interact by exchanging messages. Each node can host one or more computational *components*. In this model, a component is a self-contained hardware/software unit that interacts with its environment by exchanging messages. A timed sequence of output messages that a component produces represents its *behavior*. An unintended behavior is called a *failure*. A real-time component contains a real-time clock and is thus aware of the progression of time. When triggered, a component starts executing its predefined computations at the start instant. The communication infrastructure provides for the transport of unidirectional *messages* from a sender component to one or more receiver components within a given interval of time. Unidirectional messages create a causal chain and eliminate dependencies between senders and receivers. A message is sent at a *send instant* and received at the receiver at some later *receive instant*. The temporal properties of a message include information about the temporal order, the send instant and latency of transport. A message contains a data field that holds a data structure that is transported between components. The communication interface is agnostic about the contents of this data field.

The *state* of a real-time component represents a separation between past and future component behaviors i.e. there must be a consistent temporal order among the events of

significance to a component. Component states enable the determination of a future output solely on the basis of the future input and the present state of the system i.e. state embodies all of system history. Here, a component is a self-contained validated unit that can be used as a building block in the construction of larger systems. In order to enable a composition of a component into a distributed set of components, the principle of *composability* should be observed. A set of components are said to be composable with respect to a specified property if the system integration will not invalidate this property, once the property has been established on the subsystem level. Examples of such properties include timeliness or testability. In composable systems, the system properties follow from the properties of the individual components. This means that in a composed architecture, the introduced abstraction of a component must remain intact, even if the component becomes faulty i.e. it must be possible to replace a faulty component without any knowledge about the component internals. Note that this principle constrains the implementation of a component, because it restricts the implicit sharing of resources among components; if a shared resource fails, more than one component is affected by the failure.

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) [116] or response times [56]. Once a *timing model* of the system is realized, 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, 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 intentions i.e. informal specification

and the formal system specification. Discrepancies between these notations are called *system specification errors*. Verification is usually 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 software 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 General Analysis Methodologies

There are several methods and techniques to analyze the design of a DRE system, studying various structural and behavioral properties for correctness. Methods include prototype testing, system simulation, and formal verification.

3.1.1 Testing and Evaluation

Testing methodologies are an important part of a software development life cycle. The goal of software testing a large suites of applications is to find and fix errors prior to delivery to the end user. However, testing real-time systems is not trivial since an execution that is deemed "correct" is dependent on both logical correctness and timeliness. Rigorous testing methods are required to cover various levels of real-time system development, including both software and hardware.

Exception handling [40] tests in code evaluate performance in the presence of bad input data. Redundant hardware measures e.g. processors in the Space Shuttle [106] test fault tolerance and management algorithms. In general, testing can be classified as one of two types: Black box testing and white box testing. In black box testing [59], the tested piece of software is treated like a black box; inputs are fed to this box and the output values are recorded. Black box testing ignores *how* the inputs are transformed into outputs. By providing an exhaustive combination of inputs, the function of the software piece is evaluated. A disadvantage to this method is that unreachable pieces of code are often bypassed. Conversely, white box testing exercises all paths in a piece of software. White box testing is driven by code inspections where software is tested line-by-line e.g. code inspection in flight engine controllers to confirm appropriate control logic for each mode of operation.

White box testing, although exhaustive, is time consuming, costly and hard to evaluate or debug.

There are various levels of real-time system testing performed in the industry. Unit testing focuses on the smallest units of software such as functional blocks and modules. Unit tests are developed or generated by the software developer and usually oriented for white-box testing. In integration testing, software evaluation is performed when sets of unit-tested software modules are *integrated* into a larger program structure. Integration testing focuses on the interfaces between pretested software components. Integration testing follows the black box testing methodology where the tester is unaware or uninterested in how each software module functions. Integration tests often extend to the system level where software is incorporated with other system entities such as hardware devices. Groups of software and hardware pieces are tested for compliance and performance.

Most testing methods for software concentrate on logical correctness and are not specialized in evaluating temporal correctness. This is an essential requirement for real-time systems. Existing testing procedures need to be improved with new methods that concentrate on determining whether a system violates specific timing constraints. Such violations mean that the computation of a piece of software required too little or too much time to complete. The goal for real-time systems testing frameworks should therefore be to find testing inputs with the shortest and longest execution times, in order to check for temporal errors. There are two main ways to find test cases - manual and automatic. In manual testing, a tester uses static timing analysis to predict the test inputs which result in the best/-worst case execution times for a piece of code. When performing such analysis, the tester takes a piece of code, analyzes the set of possible control flow paths, combined with an abstract model of the hardware architecture to obtain timing bounds [91]. This is a very labor intensive technique and the behavior of the hardware is difficult to predict e.g. in modern processors that use speculative execution [43], execution time of a piece of software becomes non-deterministic [96].

In automatic testing, the testing inputs are generated by means of some algorithm. A simple form of automatic testing is random testing where the test cases are generated at random with optimization methods to improve the randomness. In software programs that contain conditionals or looping statements, the execution time is dependent on the input data. Metaheuristic search methods such as evolutionary algorithms [112] and other genetic algorithms [92, 113] are commonly used to search for execution times. The piece of code under test is then executed on the target hardware for each generated test input and the best/worst case execution times are measured. The average of such measurements are used to select the execution times and evaluate the system design.

Prototype testing involves using a prototype approximation of the real system that is similar in computational power and connectivity. This can be considered as an adequate model of a real deployment. Unlike simulations, prototyped testing provides the analysis with actual hardware on which software can be executed at real-world speeds. The effectiveness of such testing is dependent on the closeness of the chosen computing nodes to the real-world scenario. The software execution behavior realized via prototype testing provides insights about the consistency and validity of system-level specifications e.g. average trigger-to-response times. A disadvantage to prototype testing is that the software and the overall system design must be complete, or at least almost complete. Prototype testing is usually more time consuming compared to simulation-based analysis. For this reason, testing methods can never be exhaustive i.e. enforcing every possible combination of code execution traces. So, testing only shows the presence of defects in a constrained environment and does not guarantee the absence of defects. Lastly, testing encounters various challenges in concurrent systems since the results may depend on the interleaving of concurrent activities e.g. threads, which are impossible to control. Here, *results* include worst-case execution times, end-to-end response times etc. Thus, testing is not easily repeatable, especially for large concurrent systems and does not produce concrete, certifiable results.

3.1.2 Simulation

System simulation is a commonly used, often automated technique in the industry for testing control systems and algorithms [45, 57]. The TRUETIME simulation toolbox [44], based on MATLAB/Simulink [103], simulates the temporal behavior of a multi-tasking real-time kernel containing controller tasks. The controller tasks manage processes modeled as Simulink blocks. Different scheduling policies such as Priority First-in First-Out (PFIFO) or Earliest Deadline First (EDF) can be used. The execution time of these tasks are modeled as either constants, time-dependent variables or probability distributions. Accounting for low-level details such as context switching and interrupt handling effects TRUETIME is used for simulating the execution of real-time task sets. This toolbox has also been used to simulate the behavior of communication networks. In general TRUETIME can be used to simulate and analyze the effects of timing nondeterminism on control algorithms and performance. This helps develop compensation schemes that dynamically adjust control based on measured timing variations in the real-time system.

Large projects aimed at developing real-time systems employ a variety of simulation-based tools. These tools can be classified along two dimensions: scope and abstraction level. The abstraction level can be viewed from two perspectives: functional and timing models of abstraction. The scope of the simulation can be independent sub-systems or the full system as a whole. In the context of full system simulation, the abstraction level must be functionally low enough to boot and run commercial operating systems or industrial benchmarks, and temporally low enough to support modern hardware. However, going to such detailed levels of abstraction must not result in an overall simulation performance that inhibits the study of realistic workload scenarios, in terms of execution lengths and size of data. Full system simulation tools like Simics [71] support the design and testing of computer hardware and software from within a simulation framework that attempts to approximate the final application context. Simics simulates processors at the instruction-set level, supporting various models such as x86, x86-64, ARM and PowerPC architectures.

Any Simics session can be stopped to a single step where the state can be inspected. The simulation is low-level enough to access memory traffic, set breakpoins and modify the systems e.g. adding new instructions or caches. The performance results presented in [71] show the system boot simulation for a variety of operating systems e.g. x86-64 Linux boot, simulating nearly 1.3 billion instructions in 285 seconds at 4.5 MIPS. As for scalability, Simics is able to simulate upto 30 processors, with nearly 3.3 billion instructions per processor taking 40 minutes to simulate. These results are certainly impressive given the scope of the simulation and the refinement of the simulated models. In contrast to similar tools, Simics can run actual firmware and completely unmodified kernel and driver code.

Unlike formal analysis methods, simulations do not rely on rigid mathematical reasoning methods. Simulations are also not exhaustive i.e. positive results obtained from a simulation do not certify the system's performance. Simulations like Simics, although useful in exposing erratic behaviors, do no generally provide a definitive answer to system-level verification queries. This means that a lack of deadlocks in a simulation session does not mean that the system will never reach a state of deadlock. Although random variations to the expected behavior can be enforced on a simulation, such variations to state variables will never be exhaustive. vUnlike simulations, the requirements for system *verification* are strict. If a verification query for deadlock-freedom answers a "NO", then the system is guaranteed to be devoid of deadlocks. Such a guarantee cannot be provided by simulations. When simulating a system, all of the parameters governing the simulation are made explicit. This way, from the initial state of the system, the simulation is a discrete event-progressed sequence of steps following a specific trajectory. So, for small systems with a relatively minimal set of variable characteristics, multiple simulation models are typically generated and executed in parallel and the results are interpreted. System models can be refined to great levels of detail while simulating scenarios, covering various low-level details such as scheduling algorithms and communication protocols. For a small model or a

set of models, the simulation methods are automated and the results are interpreted visually. The error-detection capabilities rely on the effectiveness of the results evaluation. An advantage to using simulation methods is that the system design need not necessarily be complete i.e. sub-models of the system can be analyzed individually for correctness with some meaningful assumptions.

3.1.3 Formal Analysis Methods

The aim of formal analysis methods is to establish system correctness with mathematical vigor. Their potential has led to an increasing use by engineers of formal methods for verification of complex software and hardware systems. Formal methods are one of the highly recommended verification techniques for software development of safety critical systems according to e.g., the best practices standard of the IEC (International Electrotechnical Commission) and standards of the ESA (European Space Agency). During the last few decades, research in formal methods has led to the development of many promising verification techniques that facilitate the early detection of errors. Investigations have shown that formal verification procedures would have revealed the exposed defects in e.g. the Ariane-5 missile [67] and the Mars Pathfinder [55, 78].

Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner. Even prior to verification, accurate modeling leafs to the discovery of incompleteness, ambiguities and inconsistencies in system specifications. System models are typically accompanied by algorithms that systematically explore all states of the system model. This provides the basis for a whole range of verification techniques ranging from exhaustive exploration (model checking) to experiments with restrictive set of scenarios in the model (simulation), in reality (testing).

Both simulation and testing methods, though common, are non-exhaustive techniques

since every possible system behavior or reachable state is not analyzed and checked. For exhaustive analysis, formal verification methods have become an applied standard, especially when analyzing hardware architectures and electronic circuits [9, 20]. Formal verification aims to cover all of the potential behaviors of the system, traversing a complete state space tree for each design. In general, formal methods enable reasoning mathematically about the correctness of a system design. This work is widely used in certifying large-scale critical hardware designs and becoming increasingly common in software. However, it must be noted that formal methods have several challenging problems that limit its use in the industry. State space explosions limit the applicability of certain formal analysis techniques to classes of systems that have highly variable behaviors. State space explosion refers to a scenario where the *state space*, the tree of possible executions from a specific initial/current state, grows exponentially as a consequence of the system design. The larger the number of system components, and the larger the number of potential internal states, the larger the state space. This means, for complex composed systems, the number of state space *nodes* that must be checked against formally specified requirements can be very large. All verification methods are affected by state space explosion, leading to long analysis times and hindered practical use. Also, formal methods are usually hard and mathematically intense. Any user of formal analysis methods needs to both understand the methodologies and the internals of the tool. Industrial strength formal methods are also uncommon, protected or too ad hoc leading to a general lack of design and analysis tools that are generic and easily applicable to large domains of systems.

There are two main types of formal and verification analysis methods studied in literature: Model checking and Theorem Proving.

3.1.3.1 Model Checking

The main principle behind *Model Checking* is to analyze a system by constructing an appropriate *model* of the system. Here, the model refers to some data structure that accurately represents the structure and behavior of the system. Using this model, a model checker explores all possible system states from a known initial state and *checks* for property violations. Similar to a computer chess program that checks possible moves, a model checker examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property. Any verification using model-based techniques is only as good ad the model of the system. State-of-the-art model checkers can handle state spaces of 10^8 to 10^9 states with explicit state space enumeration. Using clever algorithms and tailored data structures, larger state spaces (10^{20} states) can be handled for specific problems. Even subtle errors that remain undiscovered using simulation, emulation testing can be potentially revealed using model checking. In academia, model checkers are commonly used to test new tree temporal logic methods, traversal algorithms, and state space reduction techniques. Academic model checkers used for rela-time systems include Spin [48], UPPAAL [65] and Kronos [117].

Typical properties that can be checked using model checking are of a qualitative nature: is the generated result correct? Will the system reach a deadlock states when this circular dependency is in effect? Model checking requires a precise statement of the properties to be examined. As with making accurate system models, this steps leads to the discovery of inconsistencies. The system model is usually automatically generated from a model description that is specified in some programming language like C or some hardware description language like Verilog of VHDL. The system model address how the system behaves and the property specification prescribes what the system should do, and what it should not do. If a state is encountered that violates the property under consideration, the model checker provides a counterexample that indicates how the model could reach the undesired state. The counterexample describes an execution path that leads from an initial system

state to a state that violates the system property. By simulating the violating scenario, useful information can be derived in order to adapt the model or the property accordingly.

Model checking has many strengths. It is a general verification approach that is applicable to a wide range of applications such as embedded systems, software and hardware design. It supports partial verification i.e. properties can be checked individually, thus allowing focus on the essential properties first. It provides diagnostic information in case a property is violated; this is useful for debugging purposes. Model checking does not require a high degree of expertise or user interaction i.e. model checkers are simply started when required. Model checking is also gaining in popularity in the industry - several hardware companies have started in-house verification labs. Model checkers can be easily integrated into existing development cycles and the learning curve is not steep. Lastly, model checking is mathematically sound and based on graph theory, data structures and logic.

Model checking also suffers from several weaknesses: Model checking is mainly appropriate for control-intensive applications and less suited for data-intensive applications since data typically ranges over infinite domains. The applicability of a model checking method is dependent on decidability - model checking is not effective on infinite-state systems. Model checkers verify a system model and not the actual system itself; any result obtained from model checking is as good as the system model. Complementary methods such as real system testing is required. Model checking suffers from state space explosion i.e the number of states needed to model the system may exceed the amount of available computer memory. Despite the development of several very effective methods to combat this problem, models of realistic systems may still be too large to fit in memory. Model checkers require some expertise in finding appropriate abstractions to obtain system models and to state system properties in logic formalisms. Lastly, model checkers are not guaranteed to yield correct results because the model checking engine may itself contain software defects.

3.1.3.2 Theorem Proving

Formal verification is accomplished by constructing a mathematical model of the computer program, hardware or software, and then using calculations to determine whether the model satisfies desired properties. This is similar to how mathematical models are used to validate the structural design of bridges, or the aerodynamic properties of a rocket. However, the appropriate mathematics for modeling computer systems is formal logic, and the calculation is accomplished by formal deduction, which has far higher computational complexity than solvers for partial differential equations used to model physical phenomena. Consequently, formal calculation often requires recourse to human guidance or to simplified models.

Formal deduction by human-guided theorem proving i.e. interactive proof checking can, in principle, verify any correct design, but doing so may require unreasonable amounts of time, effort and skill. On the other hand, model employed with automated methods, such as model checking may be so simplified that their verification does not necessarily guarantee that the property concerned will hold on the actual systems. Mechanization of formal deduction in support of verification must therefore strike a balance between the extent and the form of human guidance required, the heuristic automation provided and the convenience of the modeling supported.

Deductive reasoning techniques like theorem provers formalize the system behavioral requirements into mathematical theorems. By proving mathematical theorems, system-level requirements are verified to hold. So, while a model checker verifies the system using an abstract model, a theorem prover makes inferences about the system using strict logical reasoning. Theorem proving tools assist in the construction of such proofs. Most such tools are not powerful enough to automate the entire process of theorem proving and so many of the involved steps are invented by the user and the theorem prover fills in the mathematical gaps. Deductive and algorithmic verification tools like the Stanford Temporal Prover, STeP [16], have shown to be useful in real-time system analysis. Real-time systems are expressed

as clocked transition systems and the specifications are provided in Linear-time Temporal Logic (LTL) [36]. STeP implements verification rules and diagrams, along with decision procedures that couple with propositional and first-order reasoning to simplify verification conditions and prove them mathematically. Similar theorem provers include HOL [41], and the Prototype Verification System (PVS) [86]. There are also *proof checkers*, which unlike theorem provers, do not generate proofs but instead check already generated proofs for validity. In general, deductive methods are not fully automated and require human intervention and expertise.

Verification using theorem proving has two advantages over algorithmic methods such as model checking - theorem provers can deal with unbounded or infinite systems and can support highly expressive, yet abstract, specifications of the system and its properties. However, theorem provers currently require guidance to be presented in their terms e.g. an interactive theorem prover such as PVS [86] requires the user to suggest case splits, lemmas, and so on during the course of the proof. A non-interactive prover such as ACL2 [102] receives human guidance through the selection and ordering of the lemmas it is invited to prove. Neither of these forms of guidance seem acceptable to non-specialists. Any system designer who is not an expert with theorem proving would rather provide guidance in terms of the system design (its properties and structure), and not in terms of its proof. This is the primary argument for choosing model checking methods over theorem proving techniques. In model checking, non-specialists are required to provide a simplistic model of the system that the model checker can accept; this is the form of human guidance in model checking. But for an average system integrator, this is more acceptable and often times easier than aiding with tough mathematical proofs.

3.1.4 Static Code Analysis

In the 1970s, Stephan Johnson, then at Bell Labs., wrote Lint [54], a tool to examine C source code that had compiled without errors in order to find bugs that had escaped

the compilation step. There are many ways to detect and reduce the number of bugs in a program e.g. JUnit [73] in Java is a useful tool for writing tests. Static code analysis [70] is defined as a method of detecting errors and defects located in the source code of a program. A static code analyzer can thoroughly analyze code and make suggestions where the code should be changed based on rules defined by the user. The range of errors that a static analyzer can detect is diverse, ranging form coding defects to meeting coding standards like CERT C/C++. Such tools can also help with formatting i.e. indents, tab spaces etc. of software by ascertaining if the written code follows a standard organizational style. Static analyzers can also produce various code metrics, such as lines of code, file counts or even the number of files changed since the last build. These metrics are used to indicate the quality of the analyzed code. Advanced tools like Mathworks' Polyspace [5] offer more complicated features that implement formal methods-based approaches to confirm the absence of runtime errors.

These tools are entirely automated and analyze 100% of the source code without execution, or the use of test cases. Execution paths are analyzed by the tool, and variable ranges and concurrent data access points are known. After analysis, static code analyzers give a detailed list of errors encountered, each with a description and place of discovery. Static code analyzers also issue warnings/errors about concurrency violations, implementation defects, boundary conditions, security weaknesses, logical errors and other general defects. However, static code analyzers only allow us to argue that the code is as follows [37] :

- As compliant with software requirements as present evaluation methods and technology allows.
- That coding errors have been minimized. Static analysis does not prove that the requirements the code was developed from were correct or show that the compiled code is correct.

There are several different techniques used in static code analysis. Control flow analysis can be conducted using tools or done manually at various levels of abstraction. This is done to ensure that code is executed in the right sequence. This helps locate syntactically unreachable code blocks and highlights parts of the code e.g. loops where termination is needed. With data flow analysis, the goal is to show that no execution paths in the software exist that would access a variable not set to a value. Tools use the results of control flow analysis in conjunction with read/write access to variables. It can be a complex activity, as global variables can be accessed from anywhere. Thirdly, Information flow analysis identifies how execution of a unit piece of code creates dependencies between the inputs to and output from that code. These dependencies can then be verified against the dependencies in the specification. This analysis is often appropriate for a critical output that can be traced all the way back to the inputs of the hardware-software interface. Lastly, formal verification, also called compliance analysis, is a process of automatically proving that the program code is correct with respect to the formal specification of its requirements. All possible program executions are explored, which is not feasible by dynamic testing alone. Verification conditions can enhance compliance analysis. They consist of conditions that should be valid at the start and end of a block of code e.g pre- and post-conditions. The analysis might start with the postcondition and work backward to the start of the block. If, on reaching the start, the precondition is generated, then the block of code is provably sound. Compliance analysis essentially performs a proof of code against a low-level mathematical specification. In this respect, it is by far the most rigorous of the static analysis techniques. However, this rigor is at the expense of cost - productivity is at around five lines of code per man-day [37].

Although various forms of static code analysis offer many advantages to the system developer, they also impose certain constraints. Using these checkers restricts the language choices that may be used and the choice of data structures used within these languages. Also, analytical methods require highly skilled staff to carry out the tests and analyze the

results. It is not a complete answer for the verification of safety-critical systems. Other forms of testing are certainly required to verify certain aspects, like executing critical features. Also, dynamic aspects of the software being analyzed are difficult to model with static analysis techniques. Most automated tools also require translation to an intermediate language before they can analyze code. Automatic translators are available for some languages but not all. Lastly, multi-tasking applications software must be analyzed one task at a time. Another form of testing is required to check task interactions.

3.2 Fundamental Timing Analysis Tools

3.2.1 Cheddar Real-Time Scheduling Framework

Cheddar [104] is an Ada framework based on real-time scheduling theory that provides tools to study temporal behavior of real time applications. These applications are often associated with timing constraints such as response times, deadlines, execution rates etc. Real-time scheduling theory helps system designers to predict the timing behavior of a set of real-time tasks with scheduling simulation and feasibility tests. Scheduling simulation involves calculating the schedule for the task set within an interval and checking timing properties. Feasibility tests allow designers to study real-time tasks without computing scheduling. The authors note that in the academic community, most of the analysis tools developed do not provide both simulation-based and feasibility test-based analysis services for real-time systems. This is the primary motivation for the Cheddar project. The development of Cheddar aimed to provide a framework which implemented many of the classical real-time scheduling theory, with feasibility tests for tasks running on single processor and distributed systems with different scheduling policies and task activation patterns. For educational purposes, each result computed by Cheddar is linked with a reference equation that derived that result. Cheddar framework is also open and portable as all of the data sent to or produced by the tools are in XML format. Since feasibility tests are only known for

a few task activation patterns and scheduling policies, the framework includes a simulation engine to simulate systems with specific temporal behavior.

In Cheddar, the characteristics of real-time applications are specified by a set of processors, buffers, shared resources, messages and tasks. The simplest of task models in Cheddar is the periodic task model [68]: Each task periodically takes up the CPU for a certain run-time during which it performs some computation, aiming to complete execution before its deadline. Scheduling simulation consists of predicting for unit of time, the task to which the processor should be allocated. As the simulation progresses, the engine is capable of checking if any of the tasks in the application have missed their deadlines. Cheddar provides most of usual real-time schedulers such as Earliest Deadline First, Deadline Monotonic, Least Laxity First and POSIX schedulers. Information such as worst/best/average case response-time, blocking time, number of pre-emptions, context switches etc. can be extracted from the simulation. If the scheduling simulation takes very long to compute for a given task set, then feasibility tests can be used.

3.2.2 UPPAAL

In recent years, the use of real-time model checking has become a maturing approach for schedulability analysis - if the model checker reveals a complete lack of deadline violations, then it is guaranteed that there will be no violations in the real system execution. In this work, the software tasks, the execution platform, timing requirements, and interdependencies are mapped to a formal analysis platform and then analyzed. UPPAAL [14, 26, 65] is one such tool.

UPPAAL was developed for the design, simulation and verification of real-time systems that can be modeled as a network of Timed Automata [7], extended with integer variables and rich user-defined data types. Here, a timed automaton is a finite state machine extended with clock variables. Clock variables evaluate to real numbers and all clocks progress

synchronously. Uppaal consists of a suite of tools for verifying safety properties of real-time systems. UPPAAL models a network of timed automata using a textual language (.ta files) and is able to translate Autograph-based GUI-driven timed automata constructs into .ta representations for verification.

The UPPAAL model checker is able to check for reachability properties i.e. whether a specific combination of control-nodes and constraints on clocks and data variables are reachable from some initial configuration. . Any schedulability problem is modeled as a set of tasks competing to obtain resources. Tasks are jobs that require the usage of resources for a finite duration of time during which the job is executed and after which the task is marked as 'complete'. Constraints to this premise define specific schedulability problems. UPPAAL is capable of analyzing various types of classical schedulability problems such as Fischer' protocol, and the Train-Gate Controller.

3.2.3 TIMES

TIMES [8] has pioneered model checking methods for real-time systems, providing an expressive task model called the *Time-Triggered Architecture* (TTA). In classical scheduling theory, real-time tasks are typically modeled as a set of periodically arriving entities that perform computation. Analysis based on such a model of computation yield highly pessimistic results. In order to relax the stringent constraints on task arrival times, TIMES uses automata with timing constraints to model task arrival times, yielding a generic task model for real-time systems. Such an automaton is schedulable if there exists a strategy such that all possible sequences of events accepted by the automaton are schedulable i.e. all the associated tasks complete before their deadlines.

TIMES is capable of code generation. From a validated design model, executable code can be generated for a target platform and the code execution preserves the behavior of the model. Given a system design, TIMES also generates a scheduler pertaining to the set the application tasks, tasks constraints and arrival patterns, and adopts a scheduling policy.

Lastly, TIMES uses the UPPAAL verification engine to verify schedulability. However, so far the tool only supports single-processor scheduling with limited dependencies between tasks.

3.3 System-level Timing Analysis Methodologies

3.3.1 Petri net-based Timing Analysis for Concurrent Systems

A Petri net [90] is an abstract, formal model of information flow. 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, particularly 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 concurrence, 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 has dynamic properties that result from its execution. The execution of a Petri net is controlled by the position and movement of markers (called tokens) in the Petri net. Tokens, indicated by black dots, reside in the circles representing the places of the net. A Petri net with tokens is a marked Petri net.

Tokens are moved by the *firing* of transitions of the net. A transition must be enabled

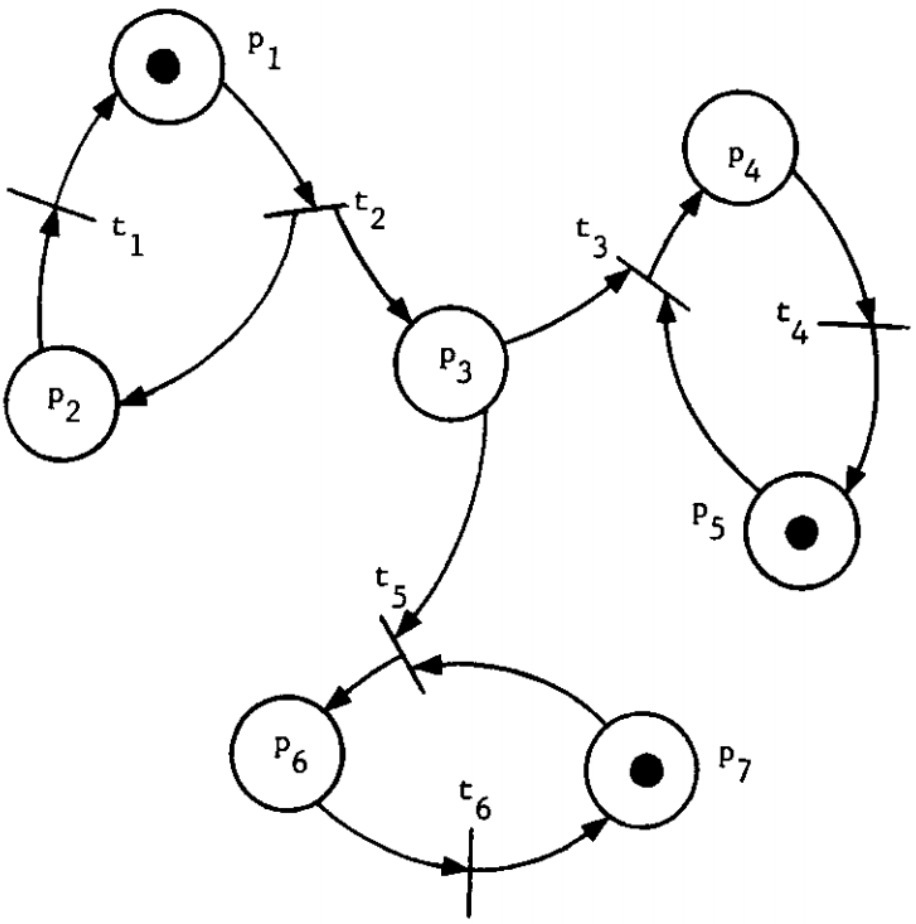

Figure 2: Sample Petri Net, reprinted from [90]

in order to fire; a transition is enabled when all of its input places have a token in them. A transition fires by removing the enabling tokens from their 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 place 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.

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. A transition can legally *fire* when all of its input place have the necessary number of tokens.

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 [27, 46], 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 [46] and timed systems [121]. For a detailed study of Petri nets and its applications, the reader is referred to standard textbooks [90, 97] and survey papers [79, 118, 122].

Petri nets have evolved through several generations from low-level Petri nets for control systems [97] to high-level Petri nets for modeling dynamic systems [53] to hierarchical and object-oriented Petri net structures [28] 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 [109], Stochastic Petri nets [12, 72] 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.

3.3.1.1 Example High-level Petri net Tool: CPN Tools

CPN Tools [95] is an open source tool for editing, simulating and analysis of Colored Petri nets (CPN), a high-level Petri net. CPN Tools supports analysis by means of explicit state space analysis and simulation. This tool can also be used to build and analyze regular

Petri nets and Timed Petri nets. CPN Tools provides a graphical user interface to edit and build large CPN, including hierarchical nets. The syntax checkers in CPN Tools run in the background and periodically check the structure of the net, along with data structures and user-defined functions.

CPN Tools currently supports two types of analysis for CPNs: simulation and state space analysis. Like many simulation software packages, CPN Tools supports single-step simulation where the Petri net is executed one transition step at a time: on an enabled transition it causes that particular transition to occur, while on a page it will cause a random, enabled transition on that particular page to occur. CPN Tools also supports executing a user-defined number of steps, where the simulation graphics will be updated after each step. A fast-forward tool will also execute a user-defined number of simulation steps, but without any graphics updates till after the last step.

CPN Tools also contains facilities for generating and analyzing full and partial state spaces for CPN. To facilitate the implementation of the state space facilities, CPN Tools uses syntactical constraints which are important for state space generation but not for simulation e.g. a state space cannot be generated unless all places and transitions have unique names, and all arcs have inscriptions. Standard state space reports can be generated automatically and saved. Such reports contain statistical information about the net e.g. boundedness properties, home properties, liveness properties and fairness properties. Querying facilities enable searching a generated state space for the presence or absence of interesting system properties.

3.3.2 Analyzing AADL models with Petri nets

Teams of researchers have, in the past, identified the need for in-depth timing analysis tools that integrate with complex system design challenges, especially in model-driven architectures [99]. Development tools like MARTE [81] and AADL [34] 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 is also concerned with model-based analysis and integration with design models. The intent here is not to define new techniques to analyze real-time systems, but instead to support them. So, MARTE supports the annotation of models with information required to perform specific types of analysis such as performance and schedulability analysis. However, the framework is more generic and intended to refine design models to best fit any required kind of analysis. Although tools exist that exercise common schedulability analysis methods like Rate Monotic Scheduling (RMA) analysis, there are very few usable tools that address the complex challenge of testing and certifying behaviors of complete, composed systems.

3.3.2.1 Translating AADL models to Petri nets:

In general MARTE provides a generic canvas to describe and analyze systems. The user is required to add domain-specific and system-specific properties and artifacts on top of the generic platform. Compared to MARTE, AADL (Architecture Analysis and Design Language) comes with a stand-alone and complete semantics that is enforced by the standard. In [99], 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 [38] 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 proposed 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 dispatching, initialization and completion; (2) Thread execution that executes thread-specific code; and (3) error management that handles potential errors. Symmetric nets are high-level Petri nets commonly used for analysis of causal properties in distributed systems, where tokens can carry data. Simple data manipulation functions are permitted allowing for powerful analysis techniques. 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 [105]. Such troubles can be alleviated by using a high-level Petri net such as Colored Petri nets 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 and cannot be easily modified for systems not modeled using AADL, especially strictly-defined component models with precise execution semantics.

3.3.3 Analyzing AADL Models with Timed Petri nets

The authors in [99], have also investigated AADL model analysis using Petri net extensions such as Timed Petri nets [98]. 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 causal 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 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 notations 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 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 Colored Petri net-based analysis work uses bounded observer places [6] that observe the system behavior for property violations and prompt completion of operations. However, this work [98] 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.

3.3.4 Mapping AADL Models to Analysis Repository - MoSaRT Framework

Modeling techniques are tailored to meet specific system requirements, simplifying design as much as possible while maintaining system integrity. Analysis techniques are developed to study scenarios of system bahavior, proving or disproving system properties based on tests. To help bridge this gap, the MoSaRT framework proposed in [85] aims at providing seamless support for real-time systems engineers during both the design of system models and analysis of system properties using several third-party tools.

This framework proposes a repository to hold analysis techniques that can be extended and enriched by analysts. The repository is used to select an analysis model that would apply for a design model. This helps point system designers in the right direction for useful analysis tests to verify properties of the system model. The analysts are expected to provide analysis methods, the elements of which are written using well-defined rules. These rules are used to add the analysis method onto the repository. These rules also help process system models to check if the analysis method is compatible with the system model.

The MoSaRT model is checked for structural correctness based on rules specified by the modeling language. Based on the violations, the initial model is tweaked and refined.

Once an analyzable model is obtained, the framework selects a suitable analysis method to apply to the model. Once the analysis method is selected, tests are used on the analyzable model to obtain useful results.

3.3.5 MAST: Modeling and Analysis Suite

MAST [39] is a modeling and analysis suite for real-time applications. MAST, still in development, aims to provide a set of tools that enable engineers and system integrators to 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 address the timing analysis results developed for both single processor [58, 68] and distributed real-time systems [88, 107].

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. 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 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 [77].

For analysis, the MAST suite includes schedulability analysis tools that use newly published research techniques such as the offset-based methods [87] that enhance analysis

results, providing less pessimistic estimates than previous results [107]. The system description is specified through an ASCII description language that serves as the input to the analysis tools.

Using UML, the real-time *view* of the DRE is described [101] 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. Some of the tools in Figure **??** like the graphical editor and results display interfaces are not available or incomplete.

3.3.5.1 MAST Modeling Methodology

MAST models a real-time system as a set of transactions. Each transaction is triggered by one or more external events, representing the set of activities that are executed by the system. Activities generate events internal to a transaction that may trigger other activities in turn. Event handlers in the model handle special events appropriately. Internal events may contain timing requirements e.g. local deadlines.

The transactional view of a MAST model represents the event flow from an external event trigger to some internal activities, each handled by an *Event Handler* and potentially causing multi-cast event objects that cause parallel activities. Here, each activity represents the execution of an operation e.g. function calls, message transmission etc. Activities are triggered by an input event and generate an output event. These are similar to a transition fire in Colored Petri nets. Each activity is executed by a *Scheduling Server*, which represents a schedulable entity e.g. processor thread, to which *Processing Resources* are allocated. These resources include the CPU and the network. Such a serving entity may be responsible for executing several activities, and thus the associated operations. Each server is assigned some parameters like scheduling policy and priority.

In general, operations represent pieces of code to be executed by the processor. MAST

modeling aims to abstract away low-level details of these pieces of code and simply describe the transactional flow along with timing properties. All operations have an execution time (worst, best and average) and scheduling parameters (priority relative to some base value).

The MAST suite is still under development and there are various missing pieces: worst-case analysis of systems with multiple-event synchronization, calculation of possible deadlocks, event-driven simulation, and a graphical editor. The schedulability analysis tools used with MAST are driven by theoretical results obtained for a large variety of real-time scenarios such as single processor, multi processor and distributed deployments, with or without scheduler preemption.

However, there is little to no sign of model checking or formal verification with MAST. The analysis tools used typically assume a specific initial state, with explicit requirements, and analyze the system based on theoretical results that tackle such requirements. Depending on the scheduling schemes, the nature of the events and the interaction medium, the execution of a transaction can, in reality, vary. Events in MAST are modeled based on timing; MAST Events can be periodic, sporadic, bursty, singular etc. MAST Events do not model the nature of the event itself. The event could block the scheduling server e.g. a synchronous remote method execution blocks the executing thread for a non-deterministic duration of time, till the serving thread finishes executing this method. These delays propagate, especially in a distributed system to various other system components causing a tree of possible executions based on even a small set of variable executions. Generating unique MAST models and analyzing them separately could solve this issue but this could lead to a potentially large set of analyzable models.

Secondly, it is unclear how easy it is to model and analyze hierarchical scheduling schemes. The modeling methods provide ways to model schedulable entities (threads) and processing resources (CPU) but not schedulers. A single CPU can have multiple layers of scheduling on top of the operating system scheduler leading to not only complicated and

interesting executions. Distributed real-time system designs are moving more and more toward component-based software development with higher layers of abstraction. It is imperative that many of the low-level schedulability analysis methods in literature be tweaked for such hierarchical systems. To be useful, the analysis tools need to be tightly integrated with the target domain: the concurrency model used by the system. The classic thread-based concurrency model (with generic synchronization primitives) is too low-level and too generic, it is hard to use, and hard to analyze. For pragmatic reasons, more restrictive, yet useful concurrency models are needed for which dedicated analysis tools can be developed.

3.3.6 Verification in AutoFocus 3

Focus is a general theory providing a model of computation based on the notion of streams and stream processing functions [18]. It is suitable to describe models of distributed, reactive systems. Based on this mathematical foundation, a tool called AutoFocus 3 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 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 [35] 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 [36] 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 [75, 76] 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

logic working 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.

CHAPTER IV

DESIGN MODEL: DISTRIBUTED MANAGED SYSTEMS (DREMS)

4.1 DREMS Component Model

Timing analysis of component-based software, as presented in this dissertation, is facilitated by the DREMS infrastructure (**D**istributed **RE**altime **M**anaged **S**ystem) [32] [83]. 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 real-time embedded systems. The infrastructure includes design-time modeling tools [30] that integrate with a well-defined and fully implemented component model [64, 83] 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 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.

The DREMS component model is based on the Component Integrated ACE ORB (CIAO) [110, 111] project. CIAO is an implementation of the OMG's Lightweight CORBA Component Model (CCM) [80]. CIAO uses the TAO [100] CORBA object request broker (ORB) as its default underlying communication middleware. With the recent standardization of connector mechanisms [82], CIAO is also able to support asynchronous messaging and the OMG Data Distribution Service (DDS) through its ports. Unlike CIAO,

the DREMS component model is not tightly coupled with the CORBA transport mechanisms. All component communication is via ports and connectors [84] enabling a variety of interaction schemes.

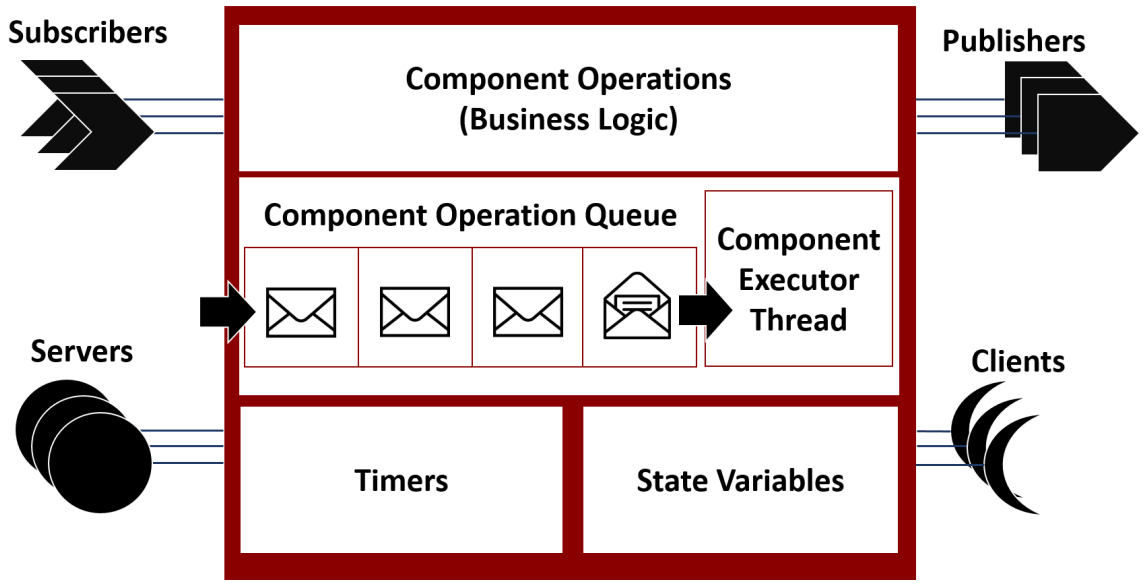

Figure 3: DREMS Component

Figure 3 presents a typical DREMS-style component. Component-based software engineering relies on the principle of assembly – large and complicated systems can be iteratively constructed by composing small reusable component building blocks. Each *component* contains a set of communication ports and 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 software. Each DREMS component supports four basic types of ports for interaction with other collaborating components: 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) [94, 108]. A component's **receptacle** specifies an

interface required by the component in order to function correctly. Using its receptacle, a component can establish connections and 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. Communication between publishers and subscribers is contingent on the compatibility of their associated topics. Publishers and Subscribers enable the OMG DDS anonymous publish/subscribe [33] style of messaging. More details on this component model can be found in [83].

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 the state variables e.g. a proportional integral differential control operation could receive the current state of dynamic variables from a *sensor* component, and using the relevant gains, calculate a new state to which an *actuator* component should progress the system. In order to service interactions with the underlying framework and with other components, every component is associated with a *message queue*. This queue holds operation requests received from the external environment i.e. other components and the underlying infrastructure. Each request is characterized by a priority and a deadline. The message queue supports various scheduling schemes including first-in first-out, earliest deadline first and fixed-priority scheduling. Depending on this choice, requests are enqueued onto the message queue for servicing.

DREMS component execution is handled by a single executor thread. 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 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 7.3.6). However, the DREMS execution semantics is still the more predictable design as it is more easily analyzable. The non-determinism in multi-threaded executions causes a tree of possible behaviors, leading to a common analysis challenge called state space explosion [66]. Keeping the operation execution to a single thread per component bounds the overall number of threads in the system leading to a more tractable analysis.

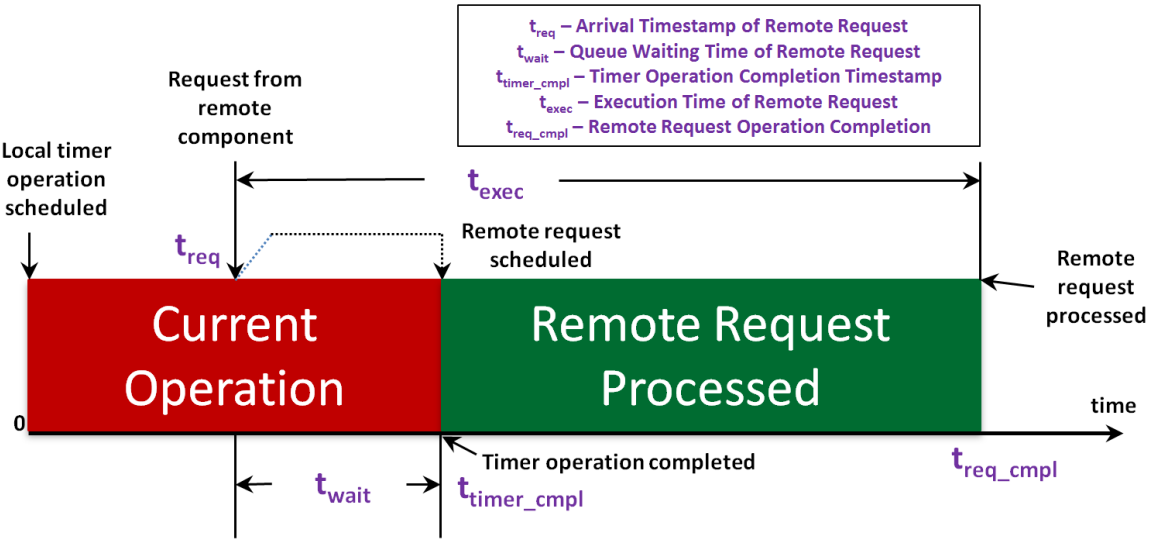

Figure 4: Component Operation Execution Semantics: This figure shows the effects of the ROSMOD component scheduling on an incoming operation request. *t_req* **represents the arrival time of a remote request.** *t_wait* **is the wait time of this request in the message queue while the current operation is still executing.** *t_timer_cmpl* **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 execution time,** *t_exec* **of this request is calculated as the difference in time stamps between** *t_req_cmpl* **and** *t_req*.

Figure 4 shows the execution semantics of a component operation executed on the component's executor thread. A simplifying assumption to describe the semantics is that this component is the only component thread executing on this CPU. Assume that at $t = 0$, this component is processing the expiry of a local timer. This operation is expected to complete at $t = t_{timer\_cmpl}$. However, at $t = t_{req}$, a service request is received from some remote component. Since the component operation scheduling is non-preemptive, regardless of the priority of this service, the request is not processed until t_{timer_cmpl}. Therefore, the request is waiting in the message queue for $t_{wait} = t_{timer\_cmpl} - t_{req}$. At $t = t_{timer\_cmpl}$, the timer operation is marked as complete and the service request is processed. The total execution time of this service operation is calculated by including the duration of the time for which this request waited in the component message queue i.e. $t_{exec} = t_{req\_cmpl} - t_{req}$.

This is a simple scenario showing how a single operation on a single component is affected by the operation scheduling semantics. The wait times of the remote request are also worsened by OS scheduling non-determinism – when multiple components are scheduled concurrently, fixed-priority real-time scheduling is enforced. If these component threads 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. Round-robin scheduling assigns a time slice to each process in equal portions and in circular order, handling all threads without priority (also known as a cyclic executive). The analysis of equal priority threads is performed by making no assumptions about the initial order i.e. any one of the equal priority threads is chosen at random and then a random circular order is established for round-robin scheduling. This causes non-determinism in the thread picking order and therefore also in the wait times of the operations executed by each thread.

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 [10] style temporal and spatial partitioning scheme.

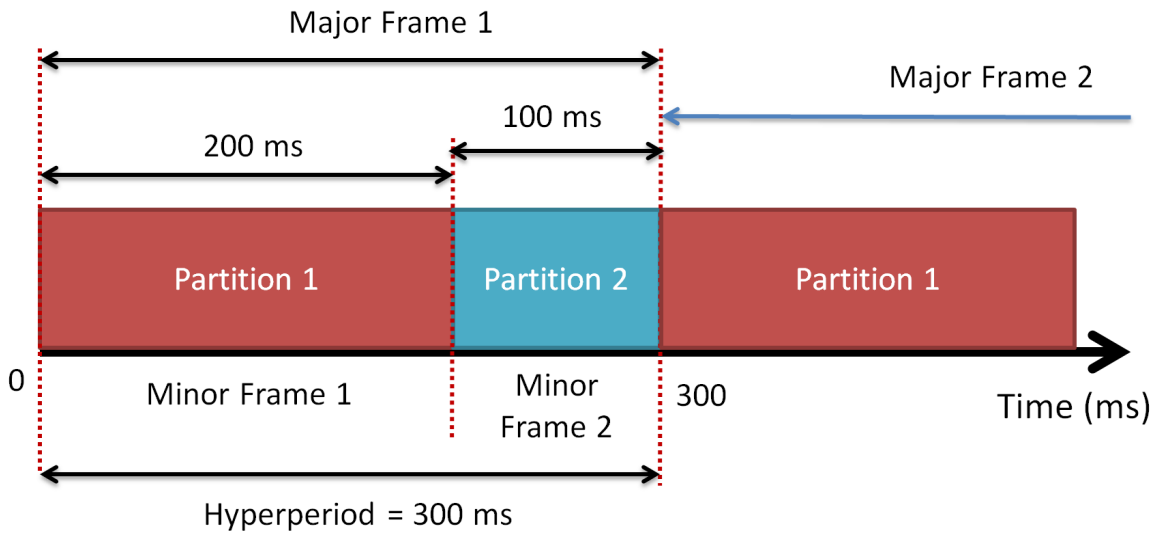

Figure 5: Sample Temporal Partition Schedule with Hyperperiod = 300 ms

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 threads 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. Each computing node in a network runs an OS scheduler, and the temporal partitions of the nodes are assumed to be synchronized, i.e. all hyperperiods start at the same time.

The DREMS component model supports non-functional properties e.g. timeliness, fault tolerance and security as an integral part of the design. Every operation on a component

is associated with a deadline. Timed triggers can be associated with operations/callbacks that dictate when and how frequently certain operations are scheduled. Deadline monitoring is invoked when an operation is allowed to execute i.e. enqueued on the component message queue. The component thread that releases the business logic execution thread monitors the deadline. If a hard deadline is reached but the operation is incomplete, then the infrastructure notifies a local fault manager and appropriate actions are taken.

Also, some long-term mission profiles e.g. space missions may involve long running computations and sensor-driven periodic calculations. Since the component execution semantics allows only one active operation to execute at a time within a component, it is possible that a ready operation is blocked for prolonged periods of time by an operation waiting on some I/O device. In such scenarios, developers can opt into using blocking I/O operations, polling mechanisms and asynchronous nonblocking I/O operations. In a blocking I/O task, the component is unavailable while the operation is running. Other components may execute in the system, but the one waiting on an I/O device is blocked. This blocking could propagate to other components and introduce significant delays. When using polling, some periodic task is scheduled that checks for the completion of I/O interaction. This leads to a potential waste of resources and decreased performance. Lastly, the component model supports asynchronous I/O, where the component triggers an I/O interaction and returns to handle other operations in the queue. The component does not block on the I/O and is notified when the I/O task completes. Such varied interaction patterns makes this component model very generic and a suitable target for our timing analysis work. The rich interactions and communication mechanisms are inspired by other common industrial component models such as CIAO [111] and ACM [31], and the execution semantics are precisely defined and implemented. A qualitative evaluation of its capabilities [83] show that although the model was designed for fractionated spacecraft, DREMS is suitable for a variety of distributed and embedded environments.

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 highly critical system threads and other low priority *best-effort* threads. Furthermore, the application threads are also subject to a temporally partitioned scheduling scheme.

System Assumptions:

1. Knowledge about the component definition, component assembly, communication ports, deployment mapping, temporal partitioning etc.

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

3. Knowledge of worst-case estimated 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.

Using this knowledge about the system, 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. 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. As described in Section 3.1.2, 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 explosions. 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. 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 3.3.1. Ordinary Petri nets have no types and no modules. Tokens are color-less dots that represent the presence or absence of some entity. The number of tokens in a place could 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), it is possible to use data types and complex data manipulations – each token has attached a data value, called the token *color*. This token color can be investigaed and modified by the occurring transitions. With CPN, 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 [95], 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 emphinscriptions, which are essentially function calls, written in Standard ML, 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 offsets.

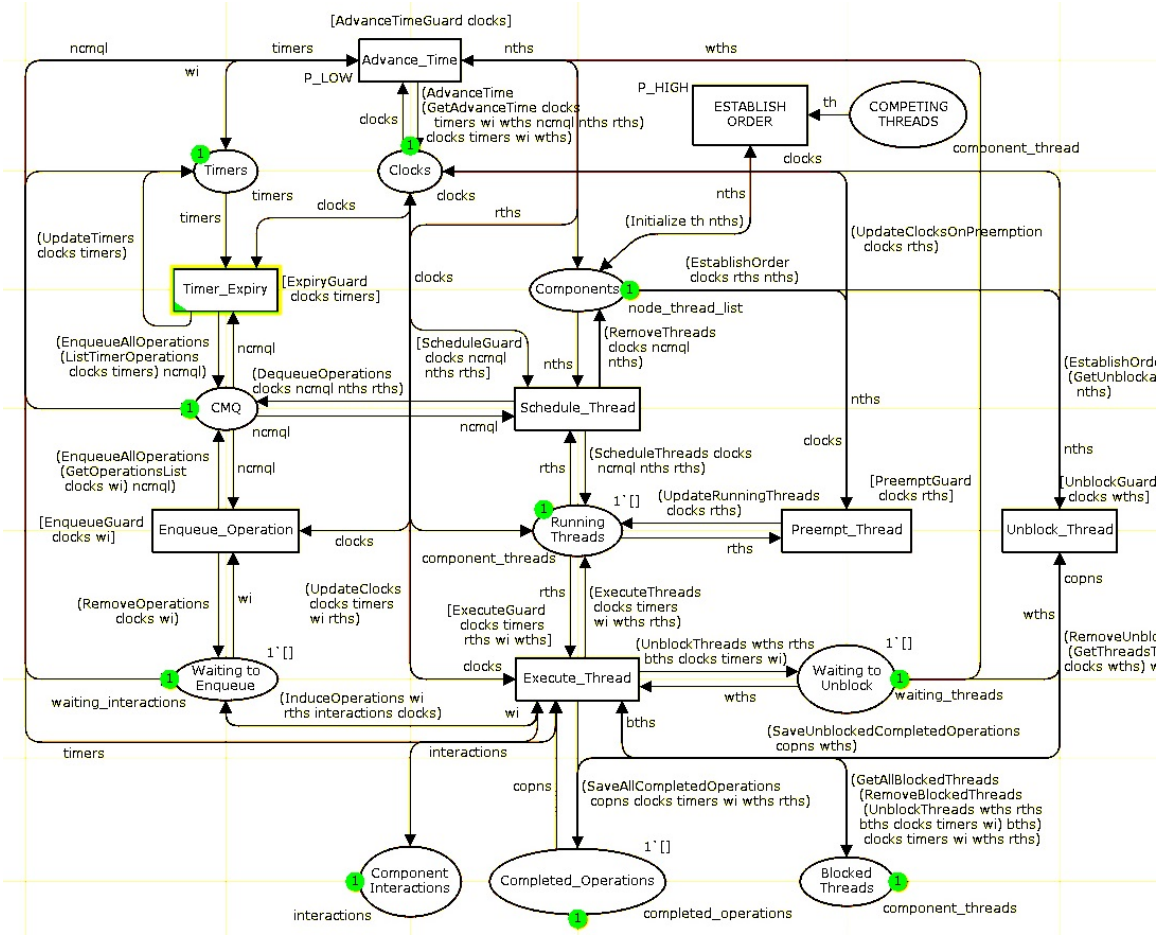

Figure 6: Colored Petri Net Analysis Model

From the design model of the system, we generate the initial CPN tokens that are injected into places in this analysis model. Using the in-built state space analysis engine, we analyze the state space of the parameterized model to compute useful system properties e.g. processor utilization, execution time plots, deadline violations etc. 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.

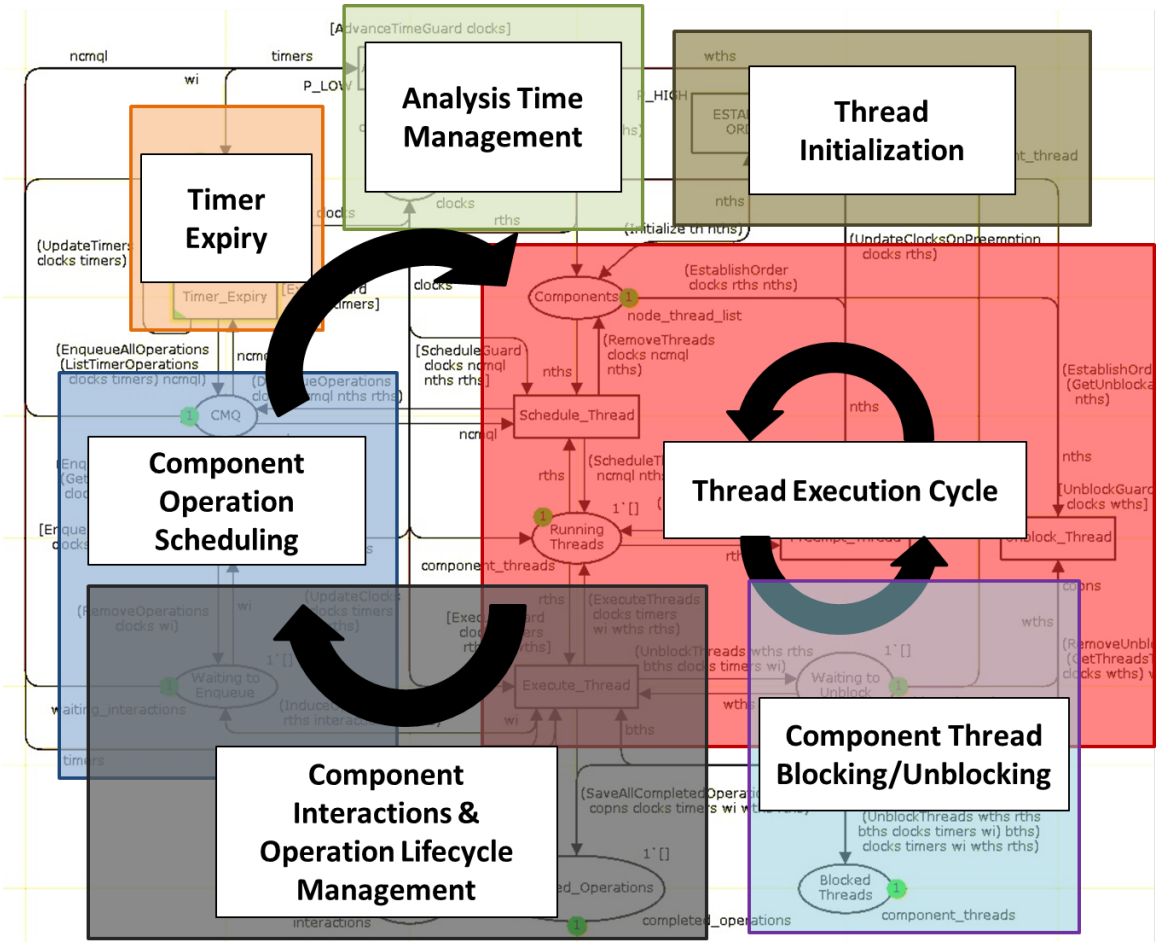

Figure 7: Analysis Model - Structural Aspects

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 multiple threads have the same priority, a round-robin (RR) scheduling is used. In order to observe and analyze this behavior, we have chosen the temporal resolution to be 100 us; a fraction of 1 clock tick of the OS scheduling quantum. 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:

$$NC = < Node_{NC},\ Value_{NC},\ TPS_{Node_{NC}} > \tag{1}$$

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.

$$TP = < Name_{TP},\ Prd_{TP},\ Dur_{TP},\ Off_{TP},\ Exec_{TP} > \tag{2}$$

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 $Exec_{TP}$.

5.2.3 Modeling Component Thread Behavior

Figure 8 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.

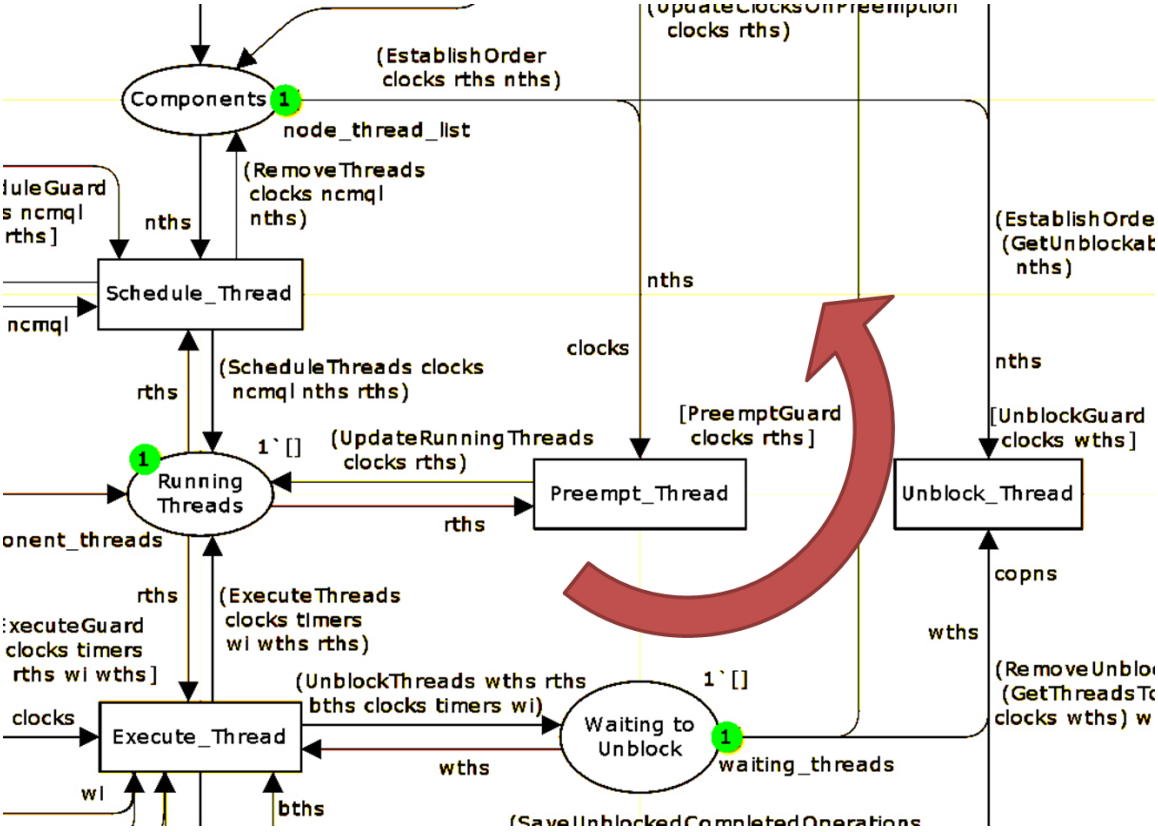

Figure 8: Component Thread Execution Cycle

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 O made on a component C_x is modeled as a Standard ML *record* of the 4-tuple:

$$O(C_x) = < ID_O,\ Prio_O,\ Dl_O,\ Steps_O > \tag{4}$$

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 analysis model also supports FIFO and EDF schemes. The completion of this enqueue implies that this operation has essentially been *scheduled* for execution. Once enqueued, if this operation does not execute and complete before its fixed deadline (Dl_O), its real-time requirements are violated.

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 9. 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 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,

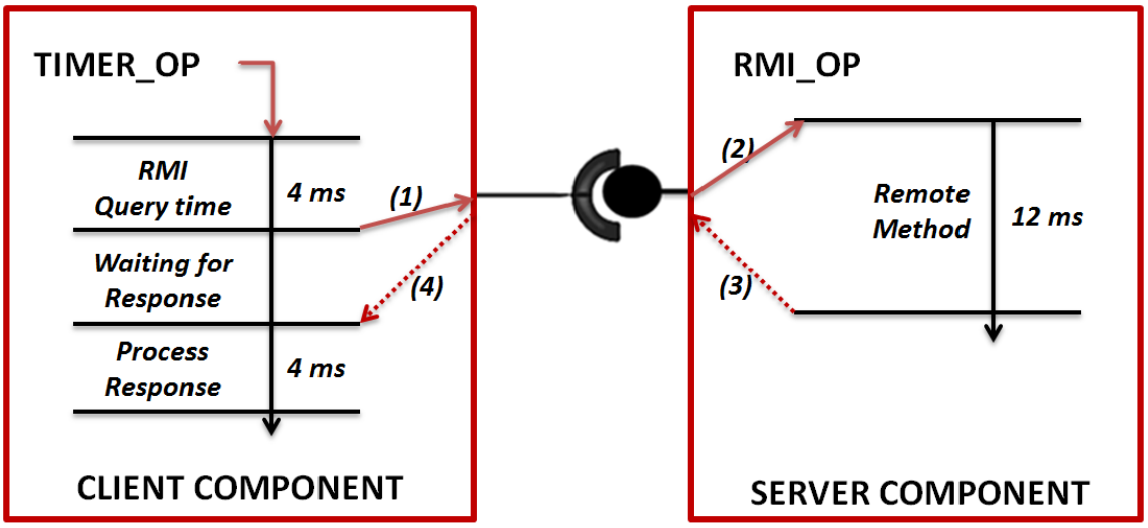

Figure 9: RMI Application

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 10.

```
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 10: 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 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 receives 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 11. 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 realized 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 11: 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 timer-triggered 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 6,12) 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:

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

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.

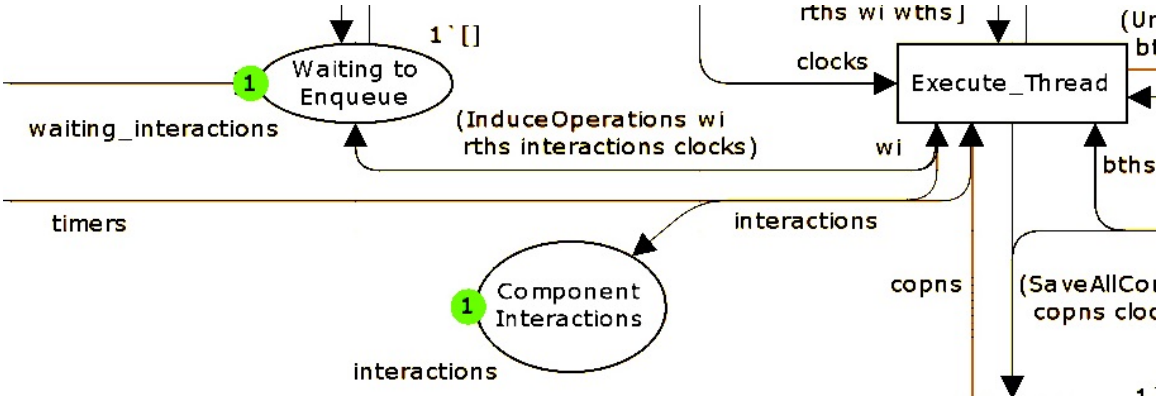

Figure 12: Operation Induction

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 13.

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

```
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 13: Operation Induction Token

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 14.

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.

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

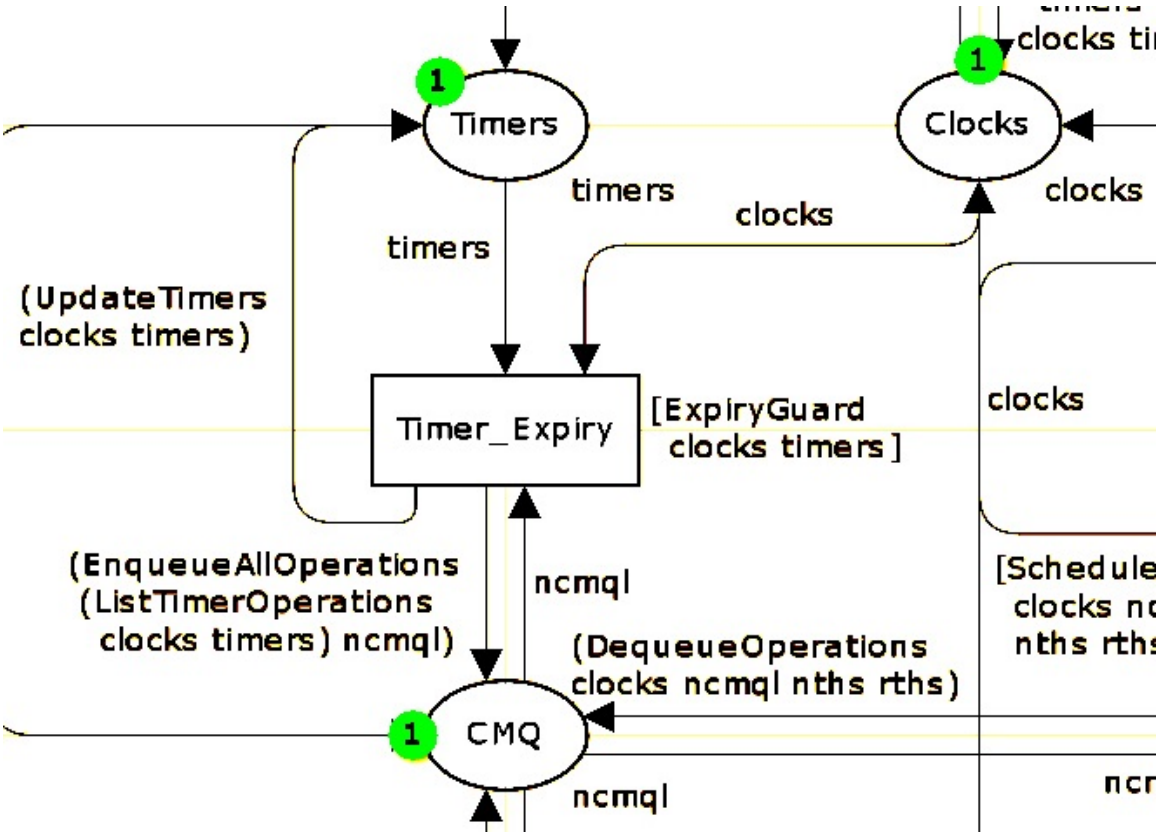

Figure 14: Timer Operations

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 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. The challenge here is to identify a metamodel grammar that would represent the 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 integrated 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 statements in the business-logic model were avoided or abstracted away. Following these rules, we designed a metamodel for the component operation business logic. Each component 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 15 shows the Extended Backus-Naur form representation of the grammar [63] 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. The arbitration is handled by the component-level scheduler. 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 an assigned worst-case execution time. We broadly classify these steps into (1) blocks of sequential code, (2) peer-to-peer synchronous and

```
(* Business Logic syntax in Extended Backus-Naur Form *)
business_logic          =   'Do', ws, operation_name, ws
                            '[', operation_priority, operation_deadline, ']', '{', { functional_step }, '};' ;
operation_name          =   ID ;
operation_priority      =   INT ;
operation_deadline      =   INT ;
functional_step         =   {sequential_code_block | rmi_call | ami_call | dds_publish | dds_pull_subscribe |
                                        dds_push_subscribe | loop} ;
sequential_code_block   =   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 '];' ;
dds publish             =   'DDS_Publish', ws, dds_port, '.', topic, '[', publish_time, '];' ;
dds pull subscribe      =   'DDS_Pull_Subscribe', ws, dds_port, '.', topic, '[', processing_time, '];' ;
dds_push_subscribe      =   'DDS_Push_Subscribe', ws, dds_port, '.', topic, '[' processing_time, '];' ;
loop                    =   'LOOP', ws, '[', count, ']', ws, '{', {functional_step}, '};' ;
receptacle_port         =   ID ;
remote_operation        =   ID ;
dds_port                =   ID ;
topic                   =   ID;
query_time              =   INT ;
processing_time         =   INT ;
publish_time            =   INT ;
count                   =   INT;
```

Figure 15: Modeling the Business Logic of Component Operations

asynchronous remote calls, (3) anonymous publish/subscribe distribution service calls, (4) blocking and non-blocking I/O interactions and (5) bounded control loops.

Notice the integration of timing properties such as worst-case function call times e.g. RMI calls (*query_time*), worst-case argument processing times (*processing_time*) and DDS publish times (*publish_time*). 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 16 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

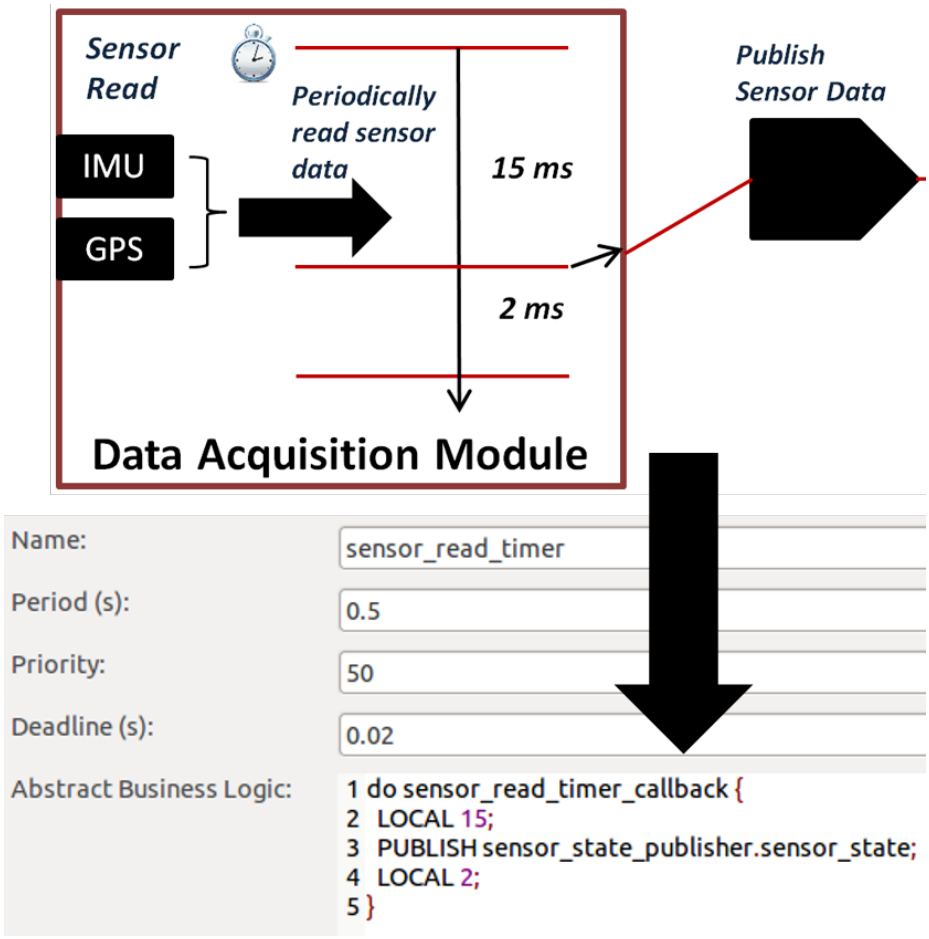

Figure 16: Sample Business Logic Model

(IMU) and GPS modules. This component packages this information and publishes sensor state as a message to all subscribers e.g. controller components. Figure 16 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 17), as described in Section 5.2.4.

```
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="DDS_PUBLISH", port="sensor_state_publisher", unblk=[], exec_time=0, duration=0},
                           {kind="LOCAL", port="LOCAL", unblk=[], exec_time=0, duration=2000}]}}]
```

Figure 17: CPN Business Logic Representation

CHAPTER VI

STATE SPACE ANALYSIS AND VERIFICATION

Consider the sample deployment shown in Figure 18. This is a 3-satellite cluster with each satellite consisting of an instance of a DREMS application. All satellites are deployed with the same component assembly and Satellite 1 is chosen as the cluster leader using a leader-election algorithm.

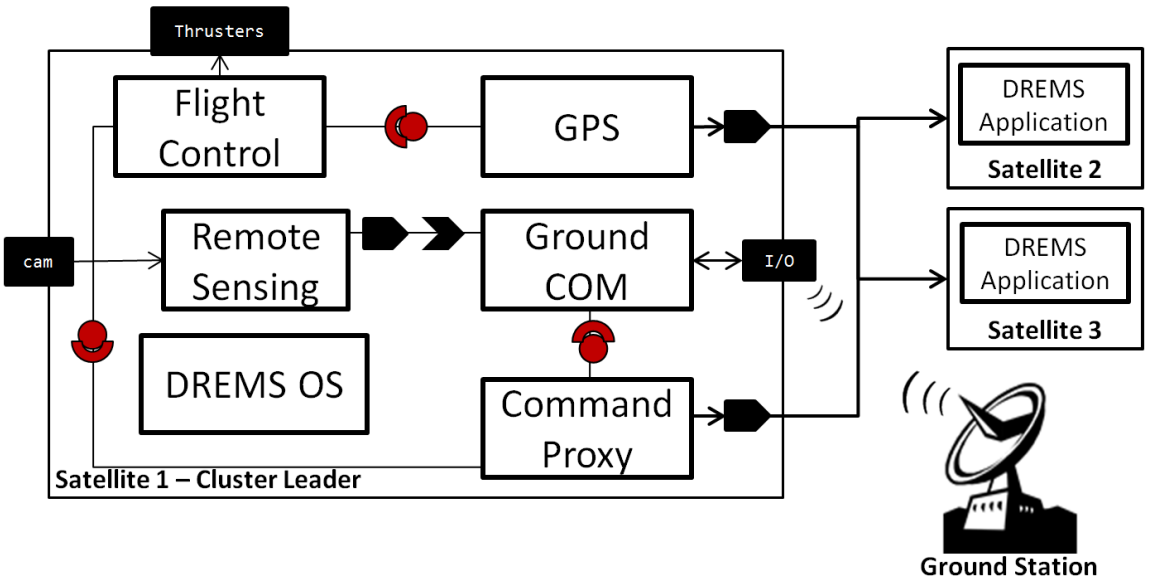

Figure 18: DREMS Application

Each satellite consists of a set of sensor-dependent critical components required for both safe flight and remote sensing tasks. These components interact with physical sensor devices such as cameras, radar altimeters etc. to receive and process data periodically and transmit to a ground receiving station. The component *Ground COM* behaves as the communication link between internal satellite components and the ground station. Periodically processed and encoded image data from the *Remote Sensing* device is received by

the Ground COM using a DDS-based *publish-subscribe* scheme upon which the data is transmitted to the ground receiver.

A more critical task in this deployment is maintaining the cluster attitude. This includes the relative position and acceleration of the satellites from each other with the satellites using a distributed but synchronized database to keep track of *state vectors* of each other. A *GPS* component periodically publishes a state vector to all satellites in the cluster. The GPS component uses its subscriber port to receive the equivalent vectors from other satellites. A *Flight Control* component responsible for the integrity of the cluster flight uses an RMI interface to access the most recent vector metrics from GPS including metrics received from other satellites to calculate and maintain the flight trajectory. The Flight Control component does not fetch GPS information until all the satellites have published on the updated sensor data.

Lastly, there are scenarios where the ground station will command the satellites in the cluster to perform a coordinated *scatter*. This is a time-triggered critical event that is transmitted to the cluster leader. On receiving such a command, the Ground COM of Satellite 1 uses an interface on the *Command Proxy* to publish this command to all satellites in the cluster. The command proxy uses a high-priority synchronous method invocation to communicate with the Flight Control component to trigger the scatter. Since the component-level scheduler is non-preemptive, the Flight Control cannot trash any operation that it would be executing at time t_S when a scatter request is received from the Command Proxy. This motivates the need for accurate modeling of component interaction semantics to obtain conservative response-time results.

The abstract business logic *steps* of the *scatter* operation in the Flight Control is modeled as shown in Figure 19. This operation is requested by the Command Proxy when a command is received from the ground station. For sake of simplicity, we have reduced this operation down to 3 distinct steps. The controller first obtains updated position vectors from the GPS component using the RMI interface. Notice the lack of a WCET for this

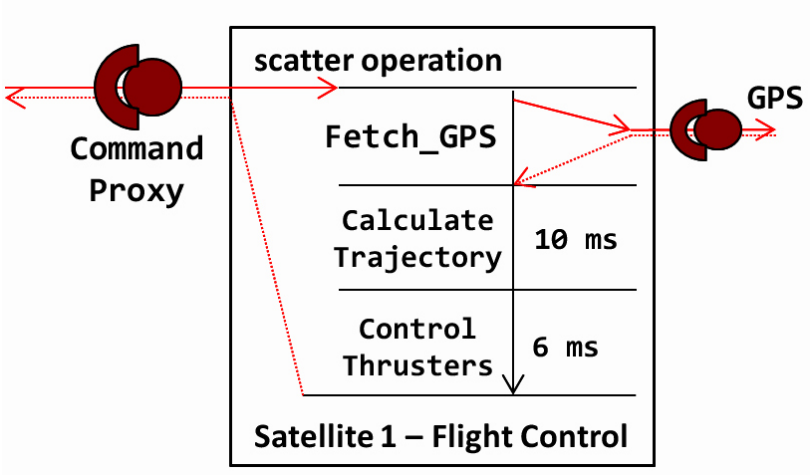

Figure 19: Scatter Operation

interaction. This is because the GPS is scheduled on a separate temporal partition and the amount of time for which the Flight Control has to wait for the updated metrics is dependent on the time stamp at which the scatter command is received and also the state of the GPS component message queue. Once these metrics are received, the Flight Control calculates a new trajectory which is also the return arguments of this operation. This trajectory is passed down to the Command Proxy which publishes the leader's updated metrics to the rest of the satellites in the cluster. Once the trajectory is calculated, each Flight Control component interacts with the thrusters and maneuvers the satellites.

The temporal partitioning schedule for this scenario consists of two broad minor frames each of length 100 msec. All satellite sensor components such as the camera and the GPS are grouped into sensor processes and assigned to the first temporal partition. This includes the image processing tasks undertaken by the Remote Sensing component, the vector publication and reception by the GPS component and the communication with the ground network.

The Flight Control is assigned to partition 2 and is guaranteed to receive the newest vector data as it waits for the GPS component to update the database of state variables. The Ground COM and the Command Proxy component threads run at SYSTEM priority and

are scheduled as and when required. In our case, we use a sporadic timer on the Ground COM component to trigger the sequence of interactions leading a scatter.

6.1 State Space Analysis

CPN Tools uses a built-in state space (SS) analysis tool to generate a bounded state space from an initialized CPN model. Exceptions caused by inconsistent token structures or incomplete arc bindings cause the tool to stop generation. However, it has been noted by the CPN Tools community that the built-in state space tool is inefficient with its search algorithm [115] and therefore we also work with the ASAP tool [114] which is an extensible platform for advanced CPN state space analysis methods including optimized reduction techniques for large-scale applications. Thirdly, we also use the ASK-CTL [21] model checking library, expressing state space properties using a CTL-style logic and exploiting strongly connected components for efficient state space searching. For smaller design models, we use observer places [6] in the CPN to *collect* tokens that represent timing anomalies such as deadline violations. This makes the state space search easier as we simply look for nodes where the observer places are not empty.

6.1.1 Bounded State Space Generation

Components in safety-critical DRE applications can be either sporadically or periodically triggered by an entity. A bounded state space generated in CPN Tools must be sufficiently large so that the lack of deadline violations or deadlocks or delayed response times *during* this interval will guarantee safe operation throughout the lifetime of the components. This is important in order to gain confidence from the obtained results while not generating an infinite state space.

6.1.2 Deadline Violations and System-wide Deadlocks

On a sufficiently large bounded state space, the analysis tool looks for specific behavioral patterns such as weakly-decreasing size of the component message queue. If requests from external components or timers pile up over time in the message queue, the responsible component is not scheduled for long enough time to be able to serve all the requests on time. This observation will also be supported by the detection of deadline violations or unusual blocking times. This is especially useful in identifying timing delays that propagate through successive hyperperiods in the temporal partition schedule.

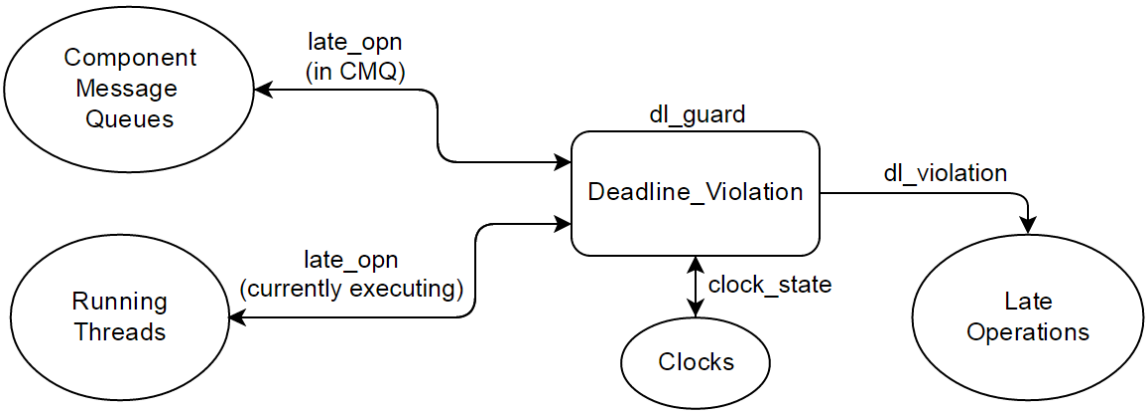

Figure 20: Deadline Violation Observer place

As mentioned earlier, our model uses an observer place for deadline violation detection, especially in small/medium sized applications. A *Deadline_Violation* (Figure 20) transition fires at any point in time when the guard *dl_guard* is satisfied and arc bindings are realized 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 in *Late Operations* (*LO*) is of the form:

$$LO = < Node_{name}, O_{name}, O_{ST}, O_{DLT} > \tag{8}$$

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.

Table 1: Component Operations on Satellite 1

Component	Operation	Dl_O (ms)	T_{NQ} (ms)	T_{DQ} (ms)	T_{FIN} (ms)	T_{EXEC} (ms)
GPS	publish_vector	10	0	0	8	8
GPS	update_dbs	18	20 (n/w delay)	20	36	16
Remote Sensing	img_process	90	0	12	80	80
Ground COM	transmit_imgs	20	80	80	95	15
Ground COM	scatter_cmd	200	120	120	315	195
Command Proxy	notify_cmd	200	132	132	142	10
Flight Control	calc_trj	45	100	100	150	50
Flight Control	scatter	200	142	150	305	163

For a large state space, using observer places is not the most efficient approach as the accumulated violation tokens themselves contribute to the state space. An easy fix to this challenge is to stop generating state space nodes after the detection of the first violation. However, if all system violations need to be recorded, then instead of using observer places, we query the state space of transition firings to find *binding elements* that indirectly suggest a violation.

System-wide deadlocks are caused by the inability of the OS schedulers (on all nodes) to schedule any component thread. This can be caused by situations where a set of executing threads are indefinitely blocked on each other because of cyclic dependencies in the interactions. Deadlocks can identified by checking the leaf nodes of the bounded state space for *dead transitions* that are unable to fire. Alternatively, the tokens in *Component Interactions* are analyzed to identify cyclic dependencies and provide warnings to possible

deadlocks. Such queries are useful in large component assemblies where mutually blocking dependencies are not immediately perceivable.

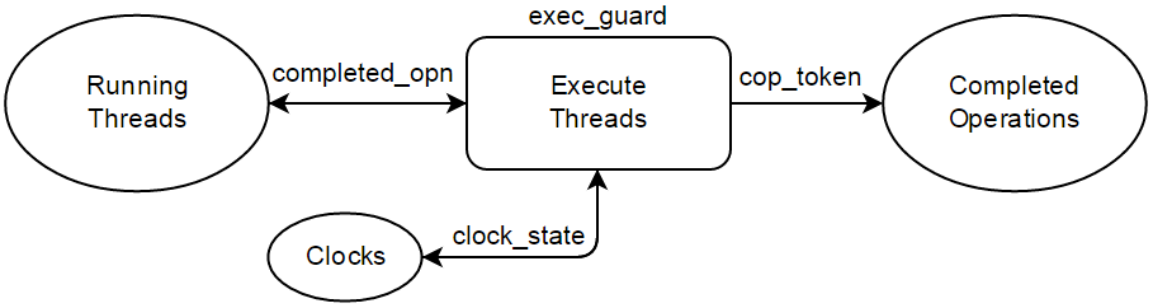

Figure 21: Response-time Analysis

6.1.3 Response-time Analysis

Response-times are measured by observing completed operations. Similar to deadline violations, an observer place *Completed Operations* (Figure 21) accumulates all operation requests completed by component threads on all nodes. The structure of this data is similar to equation 8 except we replace O_{DLT} with the end-time of the operation O_{ET}.

For a known trigger operation and an expected response/actuation, we used state space queries to identify the (1) earliest completion of the trigger operation and the (2) latest completion of a response operation. These operations are possibly running on different components, temporal partitions or nodes. To keep a check on the state space for large models, we simply observe the bindings of the transition *Execute Thread* to gather a list of completed operations, ordered by O_{ET}.

6.1.4 Search Results

Table 1 shows some worst-case execution time results from state space analysis on each component thread on Satellite 1. Each operation request is enqueued into the component

message queue at T_{NQ}. A dispatcher thread dequeues this request at T_{DQ} and schedules an operation for execution. Delays in the dequeue can be caused due to the non-preemptive nature of the component-level scheduling. Once scheduled, the component executor thread sequentially executes the steps in each operation to completion at time stamp T_{FIN} leading to an overall execution time of T_{EXEC} measured starting from T_{NQ}.

Significant worst-case network delays are assumed between interacting components that are distributed. For instance, the GPS component on each node finishes publishing sensor data at time stamp 8 ms but the GPS components on other nodes do not notice the subscription token in the message queues till $T_{NQ} = 20ms$. Also, the synchronous dependency of the Flight Control component with the GPS component is a sign of poor design as a scatter command received in partition 2 of one hyperperiod will most likely finish only a hyperperiod later as the updated GPS coordinates are queried.

6.1.5 Incomplete Designs

In order to integrate this analysis approach into early stages of component-based design, we have looked into scenarios where this work is appropriate. As mentioned in Section **??**, a component thread derives its deadlines from the operations it executes. These operations are triggered by requests from external entities with varying priorities. Identifying the optimal priority-assignment scheme for a set of component threads is non-trivial due to these variations. However, initial designs are often specified by timing requirements between system entities. Therefore, for scenarios where the developers are aware of minimum timing requirements but not thread execution orders or OS-level priorities, we have applied this approach to identify partial thread execution orders to refine incomplete designs.

Consider a sample application that consists of 6 components servicing operation requests. Components threads 1, 2, and 3 are assigned to Partition 1 and threads 4, 5, and 6 are assigned to Partition 2. The thread priorities and execution orders are unknown. Each component is triggered by a timer once every major frame, taking up to 8 ms to complete

the callback operation. Assuming a designer requires that operation 3 (handled by thread 3) must complete before 20 ms and operation 5 (handled by thread 5) must complete before 60 ms from the start of the schedule, the analysis will provide a partial thread execution order that satisfies these requirements. To facilitate this, we assign all the relevant threads equal priorities. If all the triggering timers expire at the beginning of the partition schedule, then all component threads become eligible for execution and the OS scheduler uses a non-deterministic round-robin scheduling scheme. By querying the bounded state space that encapsulates this behavior, we arrive at a partial thread scheduling order that satisfies our requirements e.g. thread 3 is scheduled first in partition 1 and thread 5 is scheduled first in partition 2.

6.1.6 Discussion

6.1.6.1 Conservative Results

Using estimates of worst-case execution time for component operations is motivated by the need to make exaggerated assumptions about the system behavior. Pessimistic estimates are a necessary requirement when verifying safety-critical DRE systems. Schedulability analysis with such assumptions should strictly provide conservative results. This means that:

- If the analysis results show the possibility of a deadline violation but the deployed system does not, the obtained result is a conservative one as it assumes worst-case behavior.

- If the analysis results do not show any timing violations but the deployed system violates response time requirements, deadlines etc., then the analysis does not provide a conservative result and has failed to verify system behavior.

In order to guarantee conservative results, the analysis must include worst-case behaviors of all system-level threads that run at higher priority than component threads and are

not necessarily modeled by the design-time tools. These threads can be grouped into a set of critical processes with approximations made to simulate the behavior of system-level threads such as (1) globally periodic CPU utilization, (2) CPU utilization for some WCET per partition etc. The best approximation is chosen based on the expected behavior of such critical processes. Best-effort processes are ignored as they always run at a priority lower than the lowest-priority component thread.

6.1.6.2 Scalability

One of the main concerns in comprehensive design-time analysis of this kind is scalability. As the determinism in the initial design increases, the number of possible behaviors and therefore the size of the state space decreases. In essence, the effort required for the analysis to be useful to the designer is dependent heavily on the initial design itself. Increasing the number of equally prioritized components will exponentially increase the number of state space nodes required to accumulate the set of behaviors that are exhibited by the components. In [**?**], we presented results showing our analysis model scaling well for medium-sized applications tested up to a 100 mixed-criticality components distributed on up to 5 computing nodes. Although these results were based on design models that made some unrealistic assumptions e.g. 100 timers expiring at the same time and triggering interactions, we have used some heuristics to reduce the generated state space and improve the performance of the search methods.

Symmetry One of the main advantages of using component-based design for complex systems is reusability. It is not unusual to deploy instances of the same application with the same component assembly on multiple computing nodes. If these applications are deployed on symmetrical nodes, the observed state space of behaviors are also symmetrical. To enable efficient search through symmetrical state spaces, we use symmetry-based state space reduction techniques [60] to improve the performance. In Figure 6, to model 100 timers, we moved from using 100 colored tokens in the *Timers* place to using a single token which is a

list consisting of a 100 elements. This way, when multiple timers expire, a single transition firing handles all the expiries across all nodes (a single state change) instead of multiple transition firings leading to multiple new states. This works well when a single instance of an application is deployed on several computing nodes with little non-determinism. The non-determinism caused by the OS-level scheduling of components and component-level operation request collisions still cause the state space to grow across symmetric nodes.

ASAP Tool The CPN Tools GUI contributes to inefficient performance of state space generation for large CPN models. This is significantly improved by using the ASAP [114] analysis tool. The tool provides for several search algorithms and state space reduction techniques such as the *sweep-line method* [22] which deletes already visited state space nodes from memory, forcing on-the-fly verification of temporal properties. One of the disadvantages of using such heuristics is the run-time cost incurred by backward-generation of states when a backtrace for a violation is required. However, if a path from the initial state to an inconsistent state is not desired, a combination of this method and the symmetry method reduces the state space and improves verification performance.

6.2 Modeling and Analysis Improvements

6.2.1 Problem Statement

The CPN analysis work presented in [64] 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 atmost 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, in 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 representing the tokens in all it's 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 statically increasing clock value. 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.

Such issues are resolved with our analysis improvements, reported in [63]. 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 or next

scheduling preempt point. This makes the system 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. This is done by maintaining the state of each computer in a *list* instead of an unordered aggregate. This approach is inspired by the symmetry method for state space reduction [60]. These improvements are evaluated in [63], where we detail our solutions with motivating examples.

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 [10] scheduling model (Figure 22).

```
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 22: 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 measure 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. We have chosen 1 msec as the quantum (corresponding to the typical 1KHz scheduler in Linux), but it can be easily changed.

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 eventually 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.

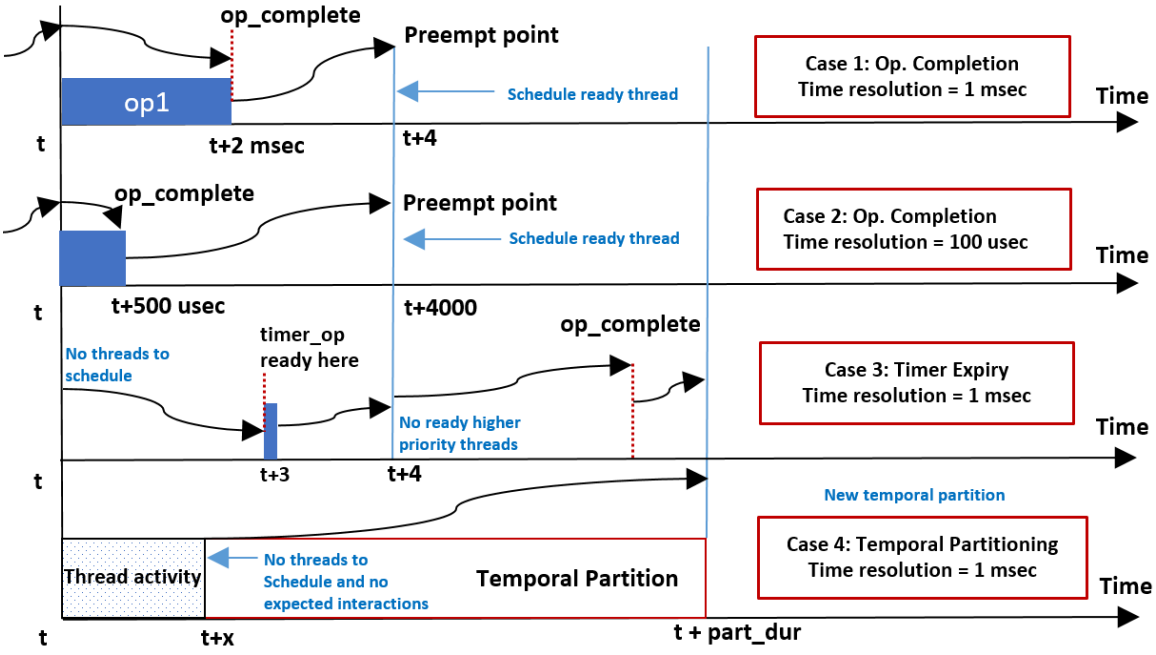

Figure 23: Dynamic Time Progression

Figure 23 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 preempt point, then there is no need to progress time in 1 msec increments as no thread can preempt this currently running thread till time t + 4 msec. Therefore, the *clock_value* in Figure 22 progresses to time t + 2 msec, 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 preempt 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 preempt point are the same, showing how the state space doesn't 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 + 3 msec, 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 preempt point. So, at time t when no threads are scheduled, the model immediately jumps to time t + 3. This scenario also shows that if the triggered operation does not complete before the preempt 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 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 [64] 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 [60].

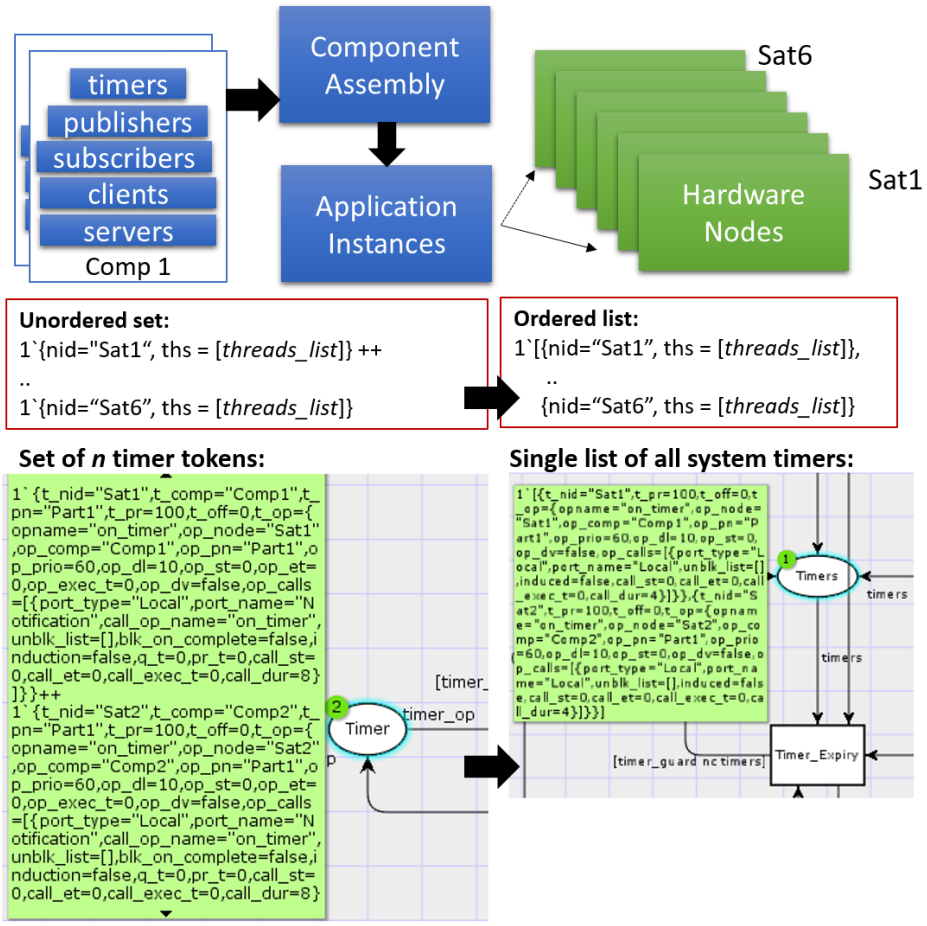

Figure 24: Structural Reductions in CPN

Figure 24 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 15). 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 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 Investigating Advanced State Space Analysis Methods

6.3.1 Problem Statement

State space analysis techniques have been successfully applied with Colored Petri Nets in a variety of practical scenarios and industrial use cases [50], [52]. The basic idea here is to compute all reachable states of the modeled concurrent system and derive a directed graph called the state space. The graph represents the tree of possible executions that the system can take from an initial state. It is possible from this directed graph to verify behavioral properties such as queue overflows, deadline violations, system-wide deadlocks and even derive counterexamples when arriving at undesired states.

Advanced state space analysis techniques arise from the need for efficient space space searching algorithms. State space analysis is challenged by time, memory, and computational power. Large state spaces require large CPU RAM and efficient search methodologies to quickly arrive at a useful result. With increasingly complexity in system designs, the number of state variables to store in memory also increases. Our problem here is to identify and apply advanced state space analysis techniques, applicable in the context of our CPN model and available as tested analysis tools that mitigate such complexities in state space analysis. This will help improve the scalability of our model and also reduce the memory footprint of the analysis.

6.3.2 Outline of Solution

The variety of CPN-specific state space reduction techniques [23], [49] developed in recent times has significantly broadened the class of systems that can be verified. In order to easily apply such techniques to our analysis model, we use the ASAP [114] analysis tool. The tool provides for several search algorithms and state space reduction techniques such as the *sweep-line method* [22] which deletes already visited state space nodes from memory, forcing on-the-fly verification of temporal properties. The main advantage of such

techniques is the amount of memory required by the analysis to verify useful properties for large models.

The sweep line method for state space reduction is used to check for important safety properties such as lack of deadlocks, timing violations etc. using user-defined model-specific queries. Practical results enumerated in [22] show improvements in time and memory requirements for generating and verifying bounded state spaces. The method relies of discarding generated states on-the-fly by performing verification checks during state space generation time. Any state that does not violate system properties can be safely deleted. Another advantage of this method compared to similar reduction methods such as bit-state hashing [47] is that a complete state space search is guaranteed.

6.3.3 Evaluation of Solution

We will evaluate these advanced state space analysis methods by comparing the obtained results with our basic state space analysis in CPN Tools. Using several criteria such as state space generation time, state space query generation, query processing time, memory usage etc., we can generate a comparison table to show the overall improvements in the analysis workflow. Advanced analysis techniques are usually accompanied by some expertise requirements that can be masked by nicely designed analysis tools. Our goal with ASAP is to use a model-driven approach to enable advanced analysis methods that does not require much expertise. With ASAP, we are able to generate verification project templates i.e. building blocks much like Simulink that are wired up together and provide an interface to the low-level analysis engine. Therefore, evaluation of this work requires both the evaluation of the advanced methods applied, and the tool used. As for the analysis methods applied, it is important to ensure that the state space tree, with all of the applied analysis heuristics, is still sufficiently probed when searching for system properties. Heuristics that enable state space analysis but only by partially checking the tree can lead the case where the state space analysis does not identify timing errors because of the incomplete search.

This will be checked with negative test cases where the design model is known to be flawed; if the results from ASAP identify injected timing errors for all test cases, then the analysis is sound.

6.3.4 Contributions

These methods were evaluated on our CPN analysis method and the results were presented in [63]. We used a large and diverse 100 component-based application for our testing. Using the CPN Tools' built-in state space analysis tool, a bounded state space of thread activity was generated. The state space generation took 36 minutes on a typical x86 laptop. We imported the same CPN analysis model onto ASAP and performed on-the-fly verification checks for lack of dead states in the analysis model for the same bounded state space. The on-the-fly verification, without any graphical interface overheads, took less than 10 minutes to compute a lack of system-wide deadlocks. It must be noted here that this improved result is due to not only because of the efficient state space search but also because of symmetry-based structural reduction discussed in the previous section.

In order to illustrate the utility of such state space reduction techniques, we consider a large-scale deployment. Figure 25 shows the generated CPN model for a domain-specific DREMS application. This is a scaled-up variant of several satellite cluster examples we have used in previous publications [32, 64]. The example consists of a group of communicating satellites hosting DREMS applications. The component assembly for this application consists of 100 interacting components distributed across 10 computing nodes, many of which are triggered by infrastructural timers. Notice in Figure 25 how there is only one token in each of the main CPN places, as described in Section 6.2.4. All of the component timers are appended to the list maintained in *Timers* place. Similarly, all node-specific clock tokens are maintained in place *Clocks*.

At time *t=0*, before the simulation is kicked off, the transition *Establish_Order* generates the powerset of thread execution orders that are possible given the configuration of

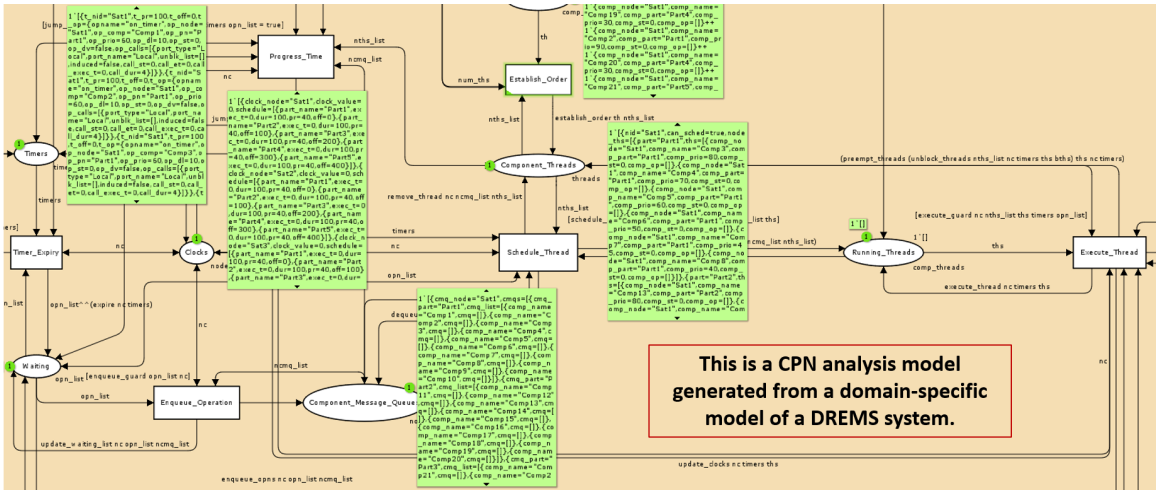

Figure 25: Generated CPN model for a Distributed Application Deployment

the clock token. This may be a potentially large set depending on the number of threads of equal priority in each partition. Once this tree of possible orders is established, the complete set of timers that are ready to expire are evaluated. Each timer expiry manifests as an operation request and each callback operation modeled using the grammar shown in Figure **??**. Once the operations are ready to execute, the highest priority component thread with a pending operation request is chosen for execution. This thread scheduling happens on all hardware nodes. When each thread executes, new interactions may occur as a consequence of the execution. For instance, if a component thread executes a timer operation in which the component publishes on a global topic, the consequence of this action would include a set of callback operation requests on all components that contain subscribers to that global topic. Lastly, all running threads are evaluated to identify the minimum amount of time that can be safely fast-forwarded in each node. If the running component threads are independent or symmetrical, then the maximum possible time progression is up to the end of the temporal partition. Note here that temporal partition in the deployment can be set to an empty list which simply removes the partitioning constraint and treats all component threads on a node as candidate threads for execution. The above sequence of transitions

repeat for as long as there is a timer expiry, a pending operation request or an unfinished component interaction.

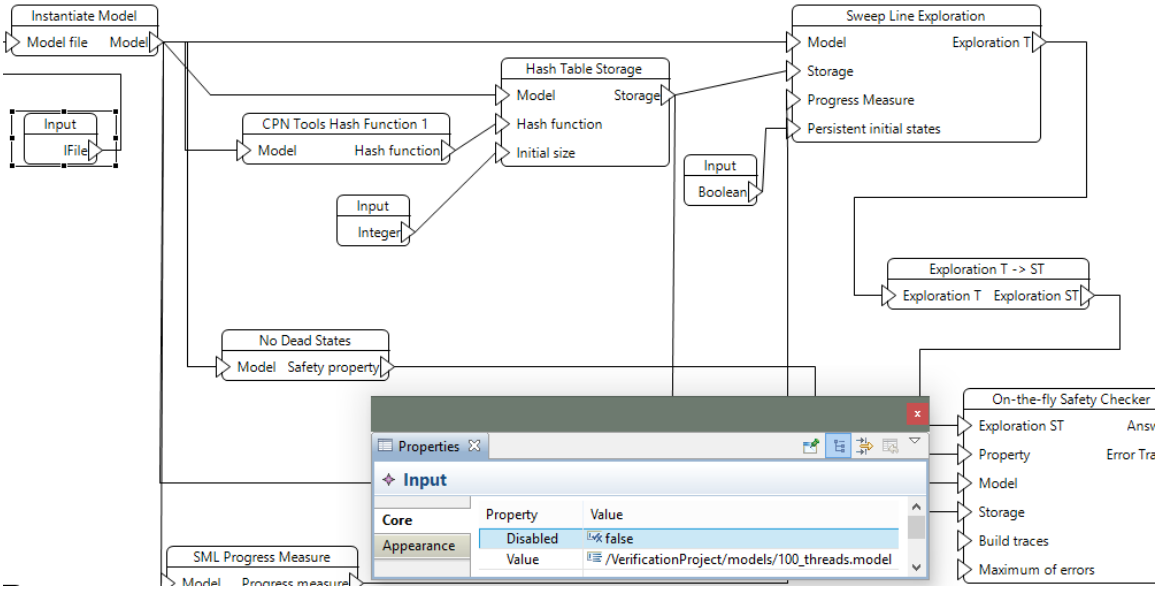

Figure 26: Sweep-Line Method

Using the CPN Tools' built-in state space analysis tool, a bounded state space was generated reaching up-to 20 hyperperiods of component thread activity. This bounded generation took 36 minutes on a typical laptop. Our goal with such an example is to evaluate the effectiveness and utility of state space reduction techniques with respect to speed and memory usage. Figure 26 shows a simple block diagram of the sweep-line method as configured in ASAP. Performing on-the-fly verification checks for lack of dead states in the analysis model, results indicate lack of system-wide deadlocks due to blocking behaviors triggered by RMI-style synchronous peer-to-peer interaction patterns. Figure 27 shows analysis results obtained from a *Verification Job* executed in the tool. Notice the on-the-fly verification taking less than 10 minutes to perform deadlock checks on this sample deployment. Using the *Palette* in ASAP, several standard ML (SML) user queries can be created to check for domain-specific properties.

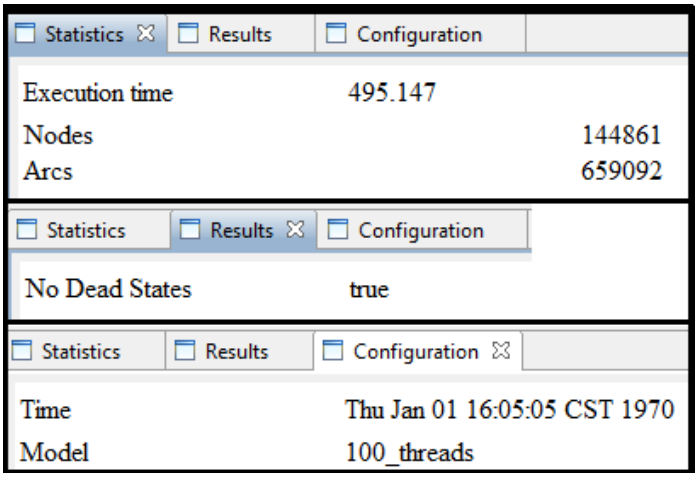

Figure 27: Dead States Checking in a Component-based application

It must be noted here that this improved result is due to not only because of the efficient state space search but also because of symmetry-based structural reduction discussed in the previous section. If not for this reduction, the state space search requirements would exponentially grow for each new hardware node added to the deployment.

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 [61] to simulate and analyze resilient cyber-physical systems consisting of 32 Beaglebone Black development boards [1]. We have chosen the light-weight ROS [93] middleware layer and implemented our ROSMOD Component model [62] on top of it. This component model provides the same execution semantics and interaction patterns as our DREMS component model [83]. Our goal with this work is 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.

7.0.1 Challenges

Experimental validation requires 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. 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 a lack of deadline violations, then the real system should follow this prediction. 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. There are

various ways to obtain the WCET values for individual operational steps but the easiest approach is to execute the design into our testbed and make accurate measurements.

WCET of component operational steps needs to be measured by having the component operation execute at real-time priority with no other component threads intervening this process. This measurement gives us a *pure execution time* of the code block. The process must be repeated for all component operations to obtain meaningful worst-case estimates that are tailored to the target platform. Obtaining the WCET values by this method is not only more realistic but also an accurate representation of the target system. Once these individual numbers are obtained, the values are plugged into the CPN through our business-logic models. Ideally, the CPN model, consisting of a composition of component operation models, when analyzed, produces results that closely resemble a real-system deployment of the component assembly. Such results would validate the modeling accuracy and the analysis results.

Distributed CPS are hard to develop hardware/software for; because the software is coupled with the hardware and the physical system, software testing and deployment may be difficult - a problem only exacerbated by distributing the system. These types of systems must be tested for performance assurances, reliability, and fail-safety. Examples of these systems include UAV/UUV systems, fractionated satellite clusters, and networks of autonomous vehicles, all of which require strict guarantees about not only the performance of the system, but also the reliability of the system. Because of the need for such strict design-time guarantees, many traditional techniques for software testing cannot be used. Cloud-based software testing may not accurately reflect the performance of the software, since many of these systems use specialized embedded computers, and furthermore does not provide the capability to easily integrate a system simulation into the software testing loop. For such systems, a closed-loop simulation testbed is necessary which can fully emulate the deployed system, including the physical characteristics of the nodes, the network characteristics of the systems, and the sensors and actuators used by the systems.

Emerging industry standards and collaborations are progressing towards component-based system development and reuse, *e.g.* AUTOSAR [11] in the automotive industry. As these systems are becoming increasingly more reliant on collections of software components which must interact, they enable more advanced features, such as better safety or performance, but as a consequence require more thorough integration testing. Comprehensive full systems integration is required for system analysis and deployment, and the development of relevant testing system architectures enables expedited iterative integration. Developing these systems iteratively, by prototyping individual components and composing them can be expensive and time consuming, so tools and techniques are needed to expedite this process. Our testbed architecture was developed to help address these issues and decrease the turn-around time for integration testing of distributed, resilient CPS.

Examples of such systems which can be prototyped and tested using this architecture are (1) autonomous cars, (2) controllers for continuous and discrete manufacturing plants, and (3) UAV swarms. Each of these systems is characterized as a distributed CPS in which embedded controllers are networked to collectively control a system through cooperation. Each subsystem or embedded computer senses and controls a specific aspect of the overall system and cooperates to achieve the overall system goal. For instance, the autonomous car's subsystems of global navigation, pedestrian detection, lane detection, engine control, brake control, and steering control all must communicate and cooperate together to achieve full car autonomy. The control algorithms for each of these subsystems must be tested with their sensors and actuators but also must be tested together with the rest of the systems. It is these types of cooperating embedded controllers which are a distinguishing feature of distributed CPS. Integration testing for these distributed CPS can be quickly and easily accomplished using hardware-in-the-loop simulation, but must accurately represent the real physical system, hardware, software, and network.

In scenarios like the automotive networked CPS, one of the main challenges with system testing is the discord between the standardized networking protocols and communication methods e.g. CAN bus, and the manufacturer-specific implementations of these methods. It is difficult to obtain public access to the implementation details for such interaction patterns and therefore pure simulation of the communication protocols using event-simulation tools is not sufficient in validating resilient application performance. The comprehensive testing for such safety-critical systems require *replicating the CPS* by using a testing infrastructure that provides similar hardware and executes the exact embedded control code that would execute on the final system.

7.1 Resilient Cyber-Physical Systems (RCPS) Testbed

7.1.1 Architecture

Cyber-Physical Systems require design-time testing and analysis before deployment. Several CPS scenarios require strict safety certification due to the mission-critical nature of the operation, e.g. flight control and automation. It is often times impossible to test control algorithms, fault tolerance procedures etc. on the real system due to both cost and hardware accessibility issues. To counter these issues, there are two principle methods in which a CPS can be tested and analyzed: (1) Construct a complete model of the CPS in a simulation environment e.g. Simulink [4] and simulate the system while accounting for run-time scenarios, (2) Establish a testing environment that can closely resemble the real CPS in both hardware and software. The problem with simulations is that it is hard to establish the network topology, emulate the application network and base processing power while running a physics simulation in the loop. Our RCPS architecture implements the latter alternative, as shown in Figure 28. We believe that a generic testing environment that uses embedded boards, programmable network switches and physics simulation machines like this, is the most suitable solution to emulate real CPS deployments.

The testbed consists of 32 *RCPS nodes*, each of which is a Beaglebone Black (BBB) [1]

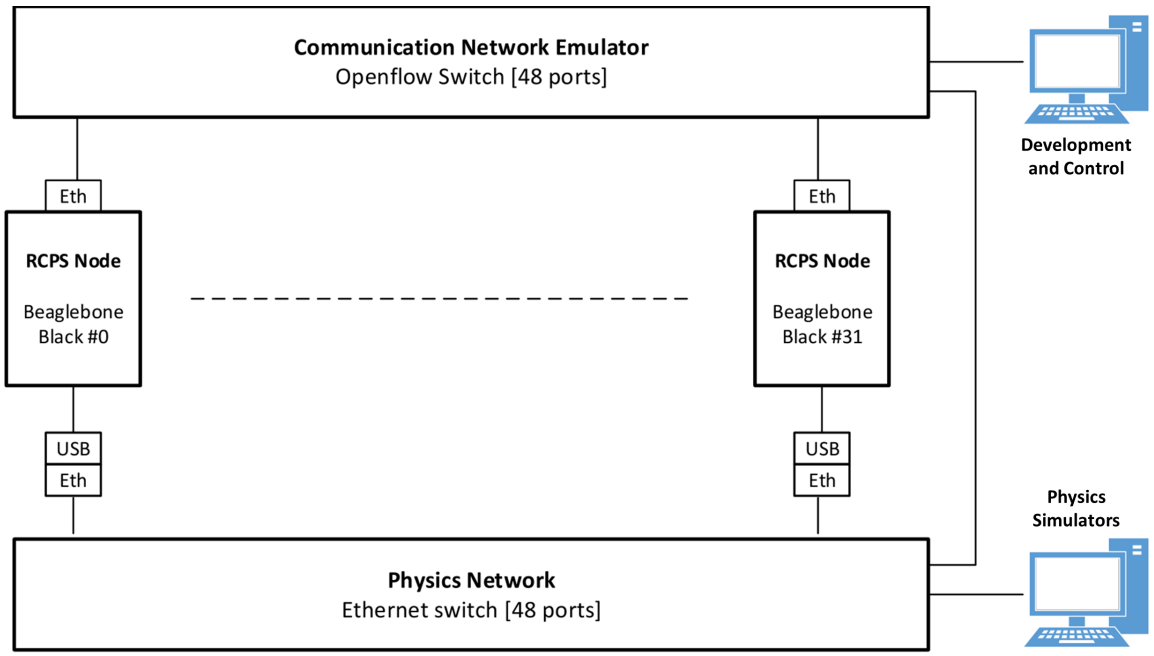

Figure 28: Testbed Architecture

development board running Linux, as shown in Figure 29. We execute a full software stack including a ROS-based middleware, called ROSMOD [62] and the DREMS component model. 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) [2] provides opportunities for nanosatellites to be deployed into space for research. CubeSats are small (4-inch long) satellites running low-power embedded boards and being prepared for interplanetary missions [3] 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.

The Gigabit Ethernet port of each BBB is connected to a *Communication Network* switch. This is a programmable OpenFlow [74] 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 throttling, enabling precise network emulation. The primary *Development and Control*

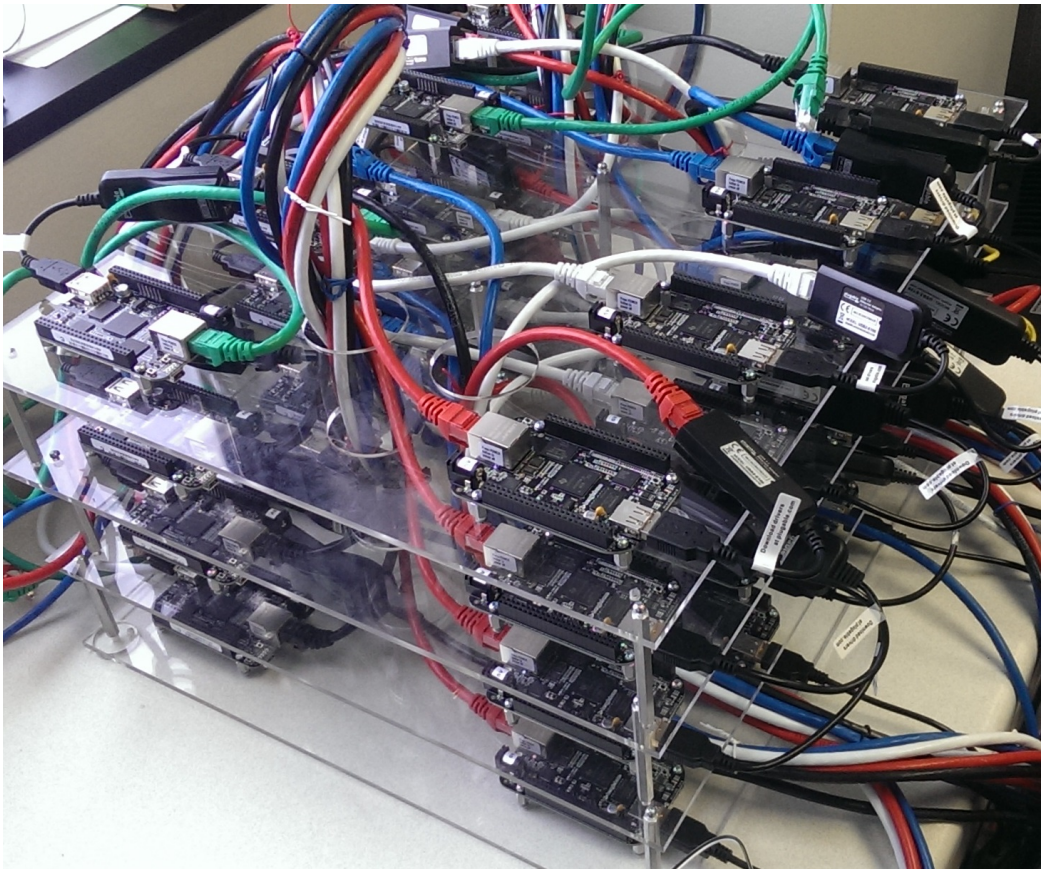

Figure 29: Beaglebone Black Boards - Custom Mounting

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 in low Earth orbit (LEO).

7.1.2 Design and Construction

The RCPS testbed was designed and constructed to be as self-contained as possible, while allowing for extensibility and modularity. The 32 BBB development boards were arranged on 4 laser-cut acrylic plates, with 8 boards on each plate, as shown in Figure 29. Holes on each plate route network and power supply cables from/to each board. With the communication network switch on top of these boards and the physics network switch on the bottom, the ensemble takes a total of 8U (rack units) of space. The primary power supply is an off-the-shelf 300 W power supply with custom cabling to fan-out power to all 32 boards in an efficient manner. By configuring the testbed in this self-contained way, we can easily maintain it, monitor it, and move it should the need arise.

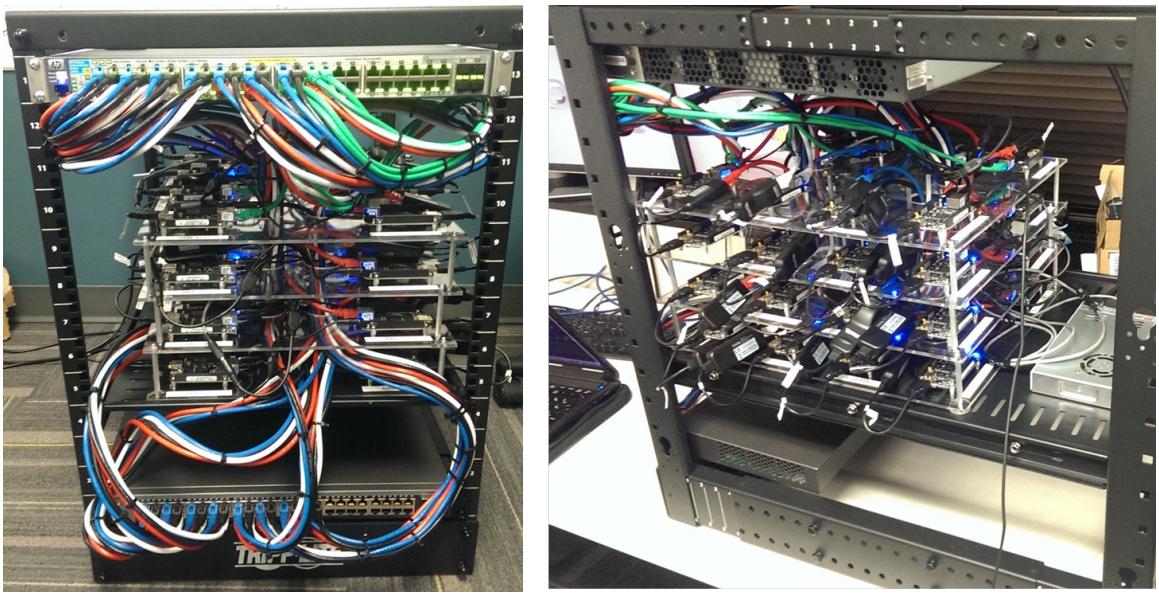

Figure 30: Constructed Testbed

Using a 13U server cabinet and standard mounting equipment, the network switches and simulation machines are mounted on either side of the development boards. Figure 30 shows the fully mounted testbed. The cabinet supports mountable base wheels which makes this setup easily portable.

7.2 ROSMOD Software Infrastructure

The software infrastructure includes our model-driven toolsuite and DREMS-style component model called ROSMOD [62], the Robot Operating System middleware [93], and component-based software applications developed for ROSMOD. The applications are cross-compiled for Beaglebone Black and the relevant processes are started at real-time priority with *SCHED_RR* linux real-time process scheduling using our ROSMOD deployment framework (Figure **??**).

7.2.0.1 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), shown in Figure 31. 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, called *plugins* 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 Meta-Model, and how they define the entities in a ROSMOD Model.

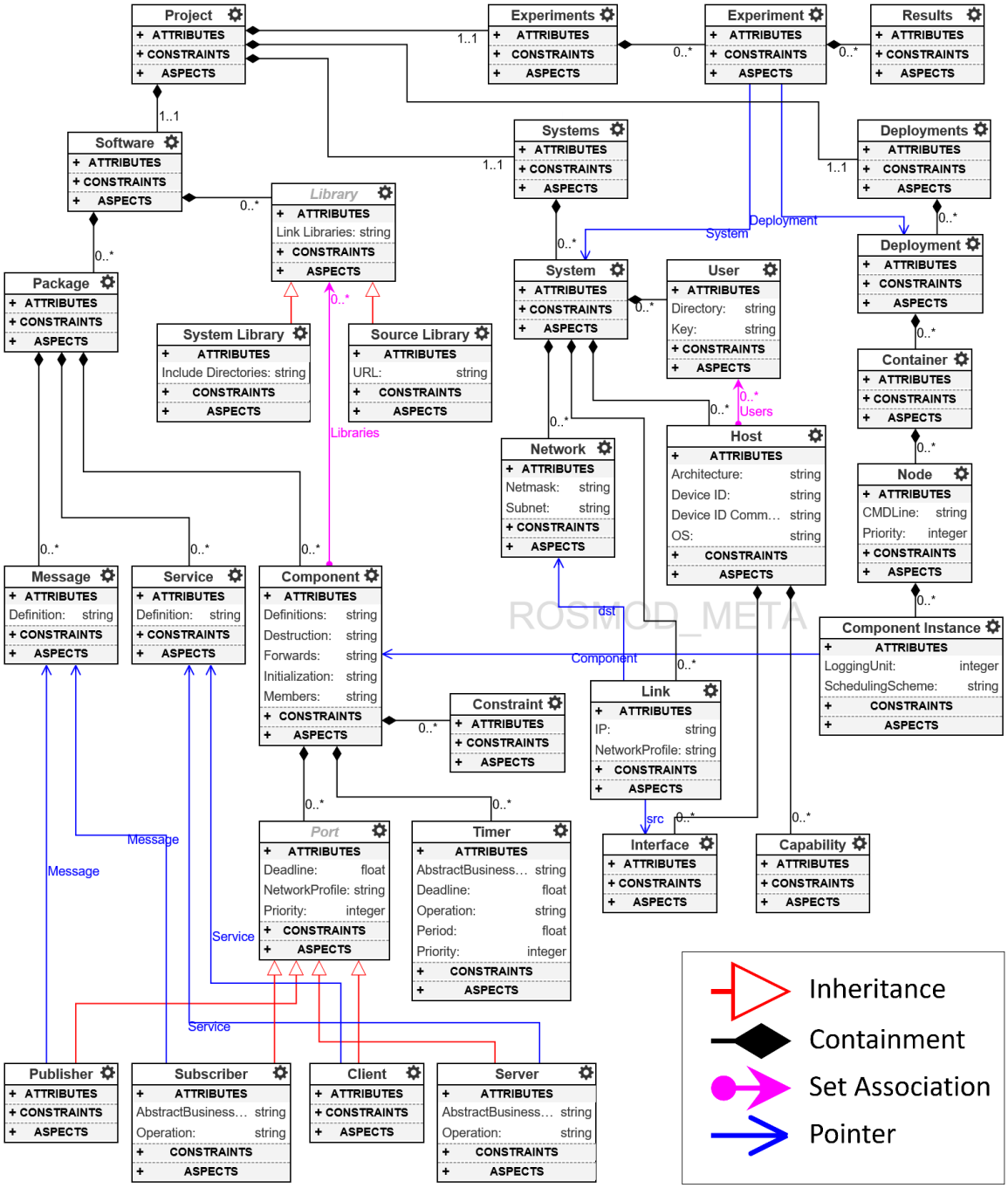

Figure 31: ROSMOD Metamodel. Containment is specified from *src* to *dst* where the source has a containment attribute *quantity*, meaning that *quantity* objects of type *src* can be contained in an object of type *dst*. Pointers are specified as a one to one mapping from source to destination, using the name of the pointer. Sets allow for pointer containment. All objects contain a *name* attribute of type *string*, not shown for clarity. Note: the meta-model is used to create the ROSMOD Modeling Language, but users do not see or interact with it; it is used to enforce proper model creation semantics.

The top-level entity of RML is a *Project*, which is shown in the upper left of Figure 31. 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 meta-model and how these concepts are integral to developing distributed CPS and rapid prototyping needs.

7.2.0.2 Software Model

The *Software* class in Figure 31 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 the ROS message description language for describing the data values used by the ROS publish-subscribe interactions. 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, as described in the component model (Figure **??**), 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 measurment sensors while operating an unmanned aerial vehicle (UAV). Lastly, components can be characterized by *Constraints*. Currently, ROSMOD constraints are a simple way to establish hardware requirements for operation e.g. fast multi-core processor, quadrature encoded pulse hardware, camera interface etc. When starting such components at runtime, ROSMOD will try to map each component to one of the available devices that satisfies all of the component's constraints. Specifying constraints for a component can be arbitrarily complex but currently we support simple feature-specific constraints e.g. *This component needs a device with atleast 12 open GPIO pins to enable motor control*. In such cases, ROSMOD will simply scan the set of devices in the hardware model and find a candidate host that *provides* such a *capability*. More about the deployment infrastructure and the mapping of constraints to capabilities is described in Section 7.2.2.

All requests received by a component, via subscribers or servers, and all timed triggers can be prioritized. The prioritization is used primarily by the component scheduler and the chosen scheduling scheme. First-in-first-out (FIFO) scheduling of received operations prioritizes based on the arrival time of the requests. Priority-based FIFO resolves conflicts between received operations by using the *Priority* attribute of the relevant ports. Similary, deadline-based schemes like the Earliest-Deadline-First (EDF) scheme uses the *deadline* of the received operation requests to resolve conflicts and operate safely. The scheduling scheme and operation priorities within a component are important choices to make since these choices directly affect the efficiency and safe operation of components. Highly critical operations need acceptable response times for robotic systems to meet quality and timing specifications. This design criteria is our primary motivation for integrating timing and performance characterization techniques into our software specification and tool suite.

The integrated performance and timing measurement system allows for the logging and visualization of system execution traces and to show the enqueue, dequeue, and completion of operations being executed by the component executor thread. As these operations trigger each other, these traces provide valuable feedback to the users about the performance and timing characteristics of the system, allowing the determination of performance bottlenecks and resource contention.

7.2.0.3 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, 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 bandwdith 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 labratory hardware for which the network is not representative of the deployed system's network.

7.2.0.4 Deployment and Experimentation

Deployment refers to the act of starting application processes on candidate hosts, where each host can provide for and satisfy all of the constraints of the processes e.g. general purpose input/output (GPIO) ports, CPU speed etc. Therefore, a deployment is a mapping between application processes and system hosts on which the application processes run. The ROSMOD Deployment Model makes this map a loose coupling to enable rapid testing and experimentation. Each Deployment consists of a set of *Containers*. Each container, conceptually, is a set of processes that need to be collocated. Containers also ensure that no process outside a container is deployed along with the container once it has been mapped to a host. Each container, therefore, contains a set of *Nodes* (ROS terminology for processes). In each node, application developers instantiate one or more components previously defined in the Software model. Following the ROSMOD component model, each such instance maps to an executor thread that executes the operations in the component's operation queue. Note that the same component can be instantiated multiple times even within the same node. Also note the container is not mapped to a specific host within the deployment model, but rather is automatically mapped to a host by the deployment infrastructure within an *Experiment*.

As shown in Figure 31, each project supports a set of *Experiments*. Each experiment has pointers to one System model and one Deployment model. The system model provides the set of available devices and the deployment model provides the set of containers, where each container contains a set of component constraints that need satisfying (the union of all the container's nodes' component constraints). ROSMOD uses these sets to find a suitable mapping and then deploys the containers on the chosen host devices, if a mapping can be found which satisfies all the constraints of all the containers. When the deployment infrastructure selects hosts from the system model for mapping, it first checks to see which of the system model's hosts are 1) reachable, 2) have valid login credentials, 3) have the correct architecture, operating system, and device ID, and 3) are not currently running any

other compilation or experiment processes from any other user. In the case that there are no available hosts in the system or the deployment's constraints cannot be satisfied, the infrastructure informs the user. Upon successful deployment of the experiment, the infrastructure automatically stores the specific mapping relevant to the deployed experiment for later management and destruction. When such experiments are stopped, ROSMOD retrieves the component logs from the hosts and displays results. We are currently working on improving our runtime monitoring features to enable real-time component execution plots and network performance measurements at runtime.

Such a loose coupling between the deployment model and the system model, along with the infrastructure automatically mapping the containers to valid, unused hosts enables the execution of the same deployment onto subsets of a large system, for instance running many instances of the same deployment as separate experiments in parallel on a large cluster of embedded devices. Additionally such a loose coupling enables redeployment onto a completely different system by simply changing the experiment's system model reference; the infrastructure will automatically verify that the new system meets the constraints of the deployment and has available hosts.

7.2.1 Software Infrastructure

The ROSMOD software infrastructure provides the user with the tools to generate source code corresponding to the software model, compile that source code to produce binaries that can run on the hosts defined in the system models, and also generate documentation for the software and associated libraries.

7.2.1.1 ROSMOD Workspace

The Software generator in ROSMOD is a plugin that produces a complete ROSMOD workspace. The application software, as defined in the model, is a collection of packages, each with messages, services, components and required libraries. When invoking the

workspace code generation, ROSMOD first generates the ROSMOD packages including C++ classes for each component, package-specific messages and services, and logging and XML parser-specific files. Then, for each external library, the URL of the library archive is used to fetch the library and inject this code as a package into the generated workspace. Assuming the library is stable and the target devices have the necessary system libraries, this generated code compiles without errors out of the box. The meta-model for the software is fully specified to include attributes for user code, the business logic for the components' operations, additional class members, etc. Because these pieces of user code are all captured within the model, they are placed into the proper sections of the generated code. For this reason, it is not necessary to alter the generated code before compilation into binaries. If the user chooses, the software infrastructure for ROSMOD will automatically determine the different types of hosts in all the current project's system models, determine which of those hosts are available that match the model specifications and will compile the source code into binaries for those architectures. The compilation is performed as a remote procedure on the host; therefore the compilation will not select hosts which are currently running other compilation or deployment processes. To enable the user to multi-task, compilation is an asynchronous non-blocking process allowing further interaction with the model or other plugins.

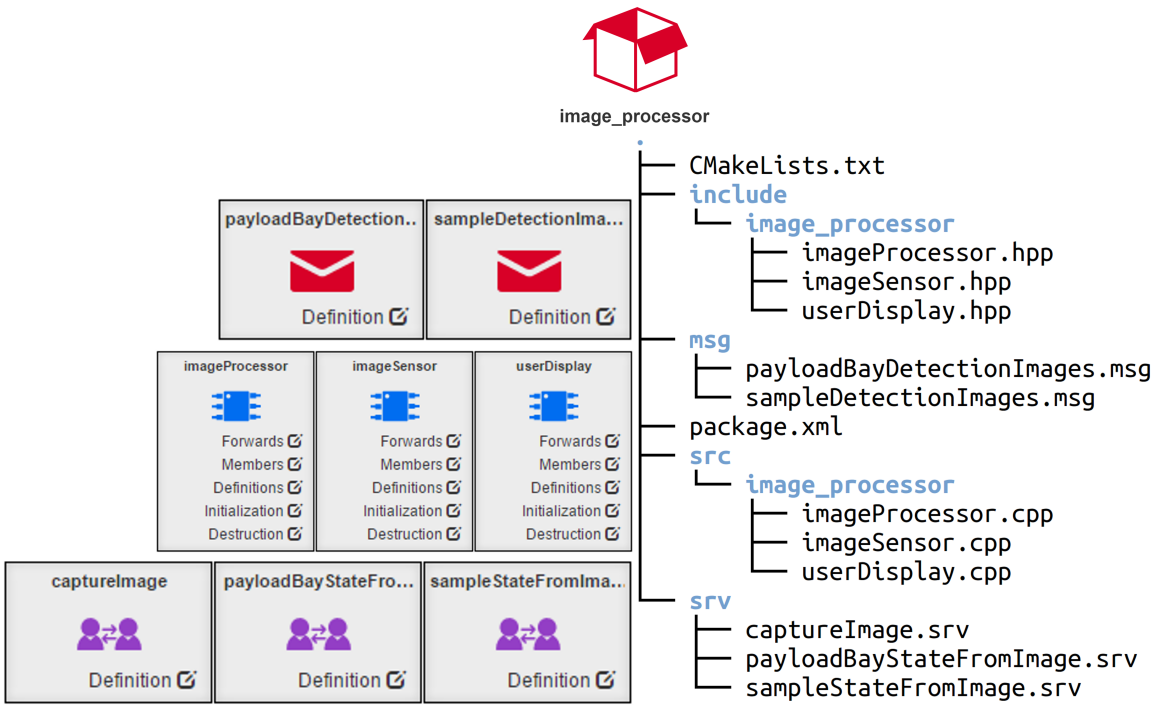

Figure 32: Workspace Code Generation. In this figure, the *image_processor* package and its children messages (red), components (blue), and services (purple) are generated into a catkin package for compilation. The msg and srv files are automatically filled out from the definitions of the messages and services, as are the header and source files for the components.

Figure 32 shows the generated code tree for an *image_processor* package. This package has three components - *userDisplay*, *imageProcessor*, *imageSensor*, and associated messages and services. For each component, ROSMOD generates a unique class that inherits from a base *Component* class and contains member objects for every component port and timer. Unlike earlier versions of ROSMOD, this generated code is not a skeleton. Rather, it includes all of the operation execution code embedded in the model. So, all changes to this business logic code are done from the model, instead of modifying a skeleton. This ensures code consistency and avoids synchronization issues between the model and the generated code. Furthermore, this enables faster training and use of the ROSMOD tool suite, since users no longer need to know into what folders and files the generated code goes. In the competition version of ROSMOD, a user would have to generate the skeleton code for

the workspace, open up the relevant files manually, find the specific places into which to place the business logic for the operations and added members, add extra library code into the generated build files manually, and then inform the infrastructure that the code was ready for compilation. This process was error prone and required detailed knowledge of what files are generated, how to integrate libraries into the build system, and other details that were specific to both ROS' package system and our component implmentation. In the current version of ROSMOD, the user is given direct access to only the code relevant for the specific object (component, message, service) they are editing. Furthermore, the library system now enables the creation and use of libraries without knowledge of how the underlying ROSMOD build system works.

7.2.2 Deployment Infrastructure

The workflow for software deployment is as shown Figure 33. 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.

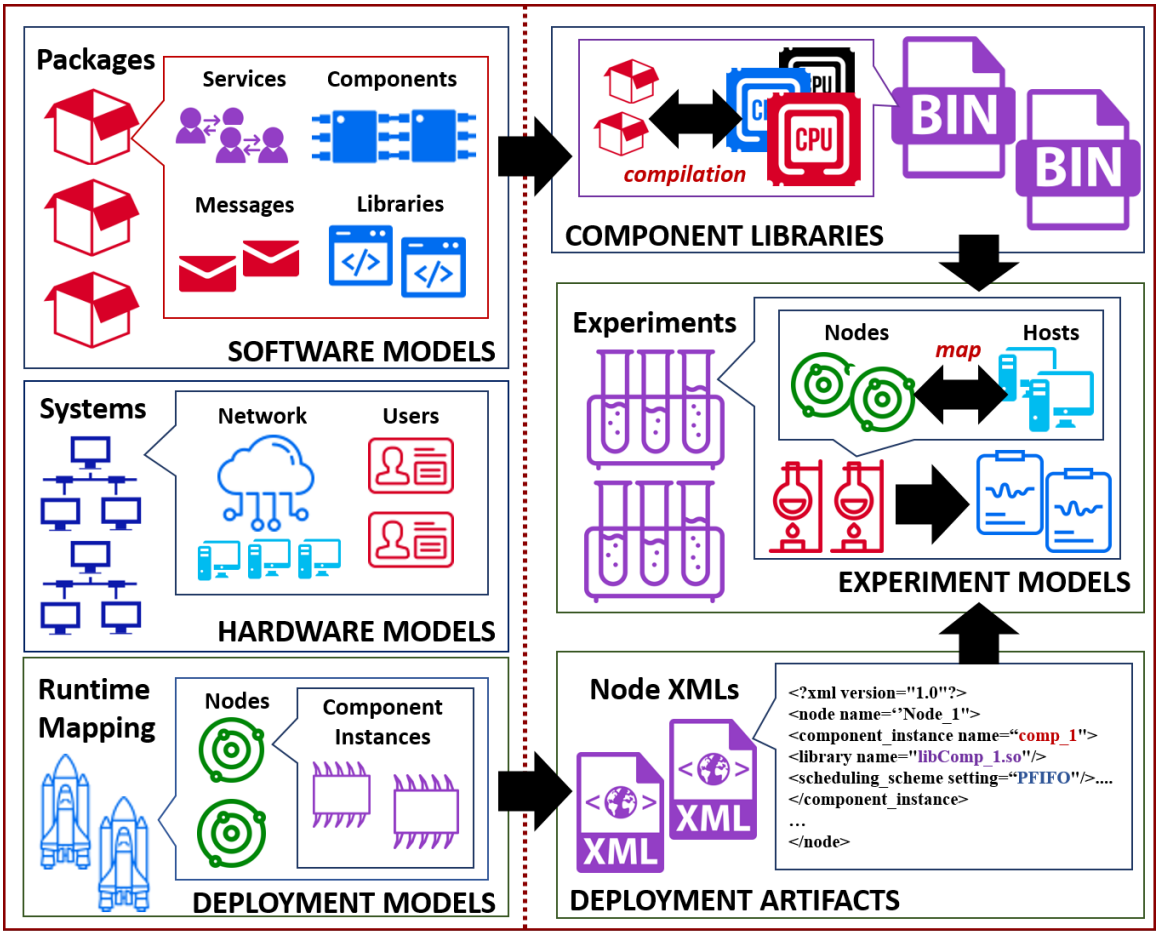

Figure 33: Software Deployment Workflow

The reason for having the components be libraries that are loaded at run-time by a node executable is 1) to help enforce separation between each component's code, 2) to shorten the compilation time (which may be quite long for embedded processors and complex models), and most importantly 3) to enable the ability for users to change the mapping of component instances to processes without the need to recompile any of the code. Because ROSMOD is focused on the rapid development and deployment of reusable software

components, we designed the infrastructure to allow users to experiment with which components are collocated within processes dynamically and rapidly iterate through several scenarios and experiments without having to wait for code compilations between experiments. Finally, such a component-based deployment framework enables unit testing and integration testing in a very well-defined manner for the software components. If an error occurs during the execution of an application on a system, the user can easily and quickly break the deployment down into sub-deployments for unit-testing of only the relevant components to determine the source of the error.

In the above architecture, the deployment needs three primary ingredients: (1) the generic node executable, (2) dynamically loadable component libraries, and (3) an XML file for each ROS node in the deployment model. For each new node added to the deployment model, by merely regenerating the XML files, we can establish a new deployment. The ROS workspace is rebuilt only if new component definitions are added to the Software Model. This architecture not only accelerates the development process but also ensures a separation between the Software Model (i.e. the application structure) and deployment-specific concerns e.g. component instantiation inside ROS nodes.

When the user has selected an experiment to run, the deployment infrastructure first determines whether the selected deployment can execute on the selected system. Like the software infrastructure described above, the deployment infrastructure queries the selected system to validate that the system is reachable, conforms to the model, and has available hosts for deployment (i.e. they are not currently running any compilation or deployment processes). Once those available hosts have been determined, the infrastructure attempts to map the deployment's containers to the available hosts based on the constraints and capabilities of the two sets. If the constraints cannot be satisfied by the capabilities of the available hosts, the user is informed and the deployment of the experiment is halted. If the capabilities of the hosts do satisfy the constraints of the containers and their associated components, the deployment infrastructure generates the required XML files for the

deployment and copies the XML files and the binaries over to the selected hosts, before starting the relevant processes. Finally, if all of those steps are successful, the infrastructure saves the mapping into the model for user inspection and for later teardown of the experiment.

When the user chooses to end a currently running experiment, the deployment infrastructure verifies that the experiment is still running before stopping the associated processes, copying the relevant log files back, cleaning up the deployment artifacts from the hosts, and removing the saved experiment mapping from the model.

7.3 Evaluation of Timing Analysis Results

Experimental validation should 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. 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. We have previously designed [63] a business-logic modeling grammar that 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 9. 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 9, 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 the number of factors responsible for this delay are numerous 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 CPN model must capture and account for such delay-causing factors.

WCET of component operational steps needs to be measured by having the component operation execute at real-time priority with no other component threads intervening this process. This measurement gives us a *pure execution time* of the code block. The process must be repeated for all component operations to obtain meaningful worst-case estimates that are tailored to the target platform. Obtaining the WCET values by this method is not only more realistic but also an accurate representation of the target system. Once these individual numbers are obtained, the values are plugged into the CPN through our business-logic models.

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 [63]. As mentioned earlier, in all of our tests, we use the ROS [93] middleware and our ROSMOD [62] component model.

7.3.1 Understanding the CPN Analysis Plots

By performing state space analysis, we are analyzing a bounded tree of possibilities. A tree of execution traces provided by the non-deterministic execution of our timing analysis model. 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 execution time behavior of all components in this trace. This pattern is followed in all of the following plots. Here, we discuss how the plot is made and how it must be interpreted.

7.3.2 Client-Server Interactions

As shown in Figure 9, a simple client server example involves a periodically triggered client component that fetches data from a remote server. Figure 34 shows our experimental trace of a simple distributed client-server sample. The client (client_timer_operation) is triggered every 500 ms, and performs floating-point calculations in a loop requiring the services of a remote operation. The server (Power_operation) periodically receives this operation request and responds to it, taking about 1.2s to complete each operation instance. In this experiment, these component threads are running at high uninterrupted real-time priorities.

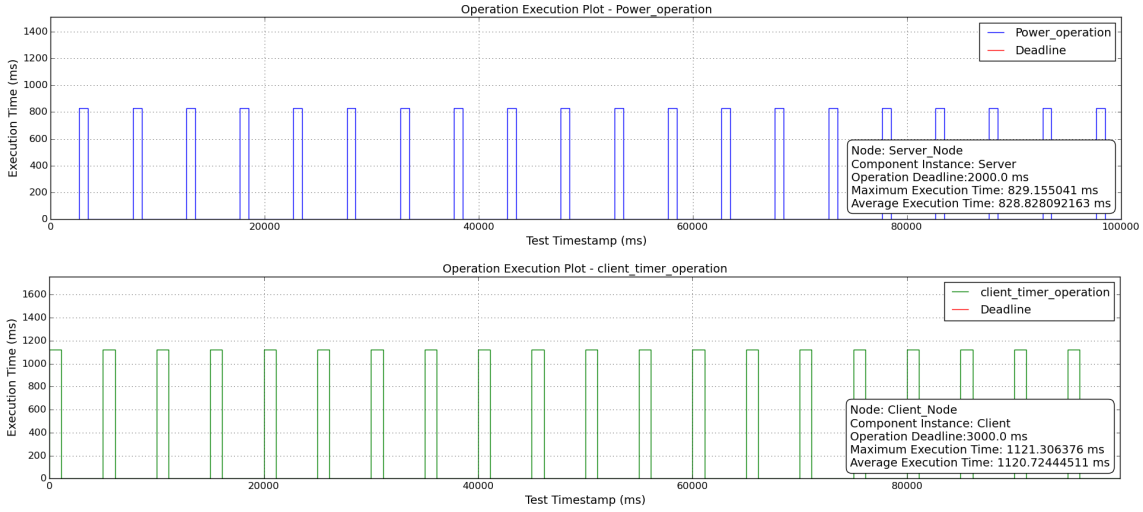

Figure 34: Experimental Observation: Client-Server Interactions

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.

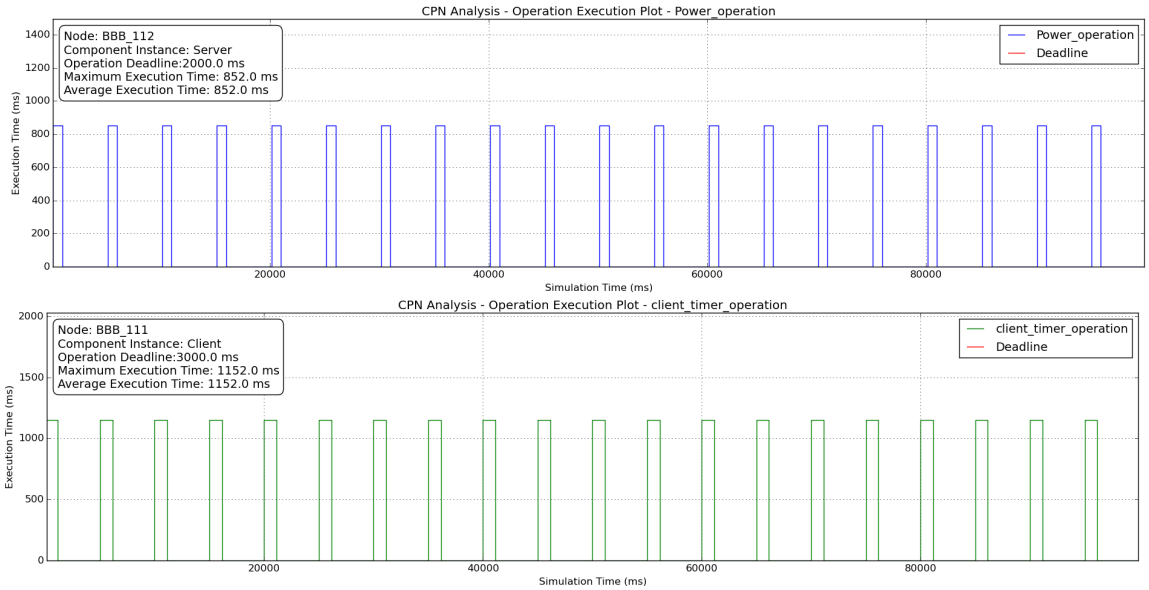

Figure 35: CPN Analysis Results: Client-Server Interactions

7.3.2.1 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 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.

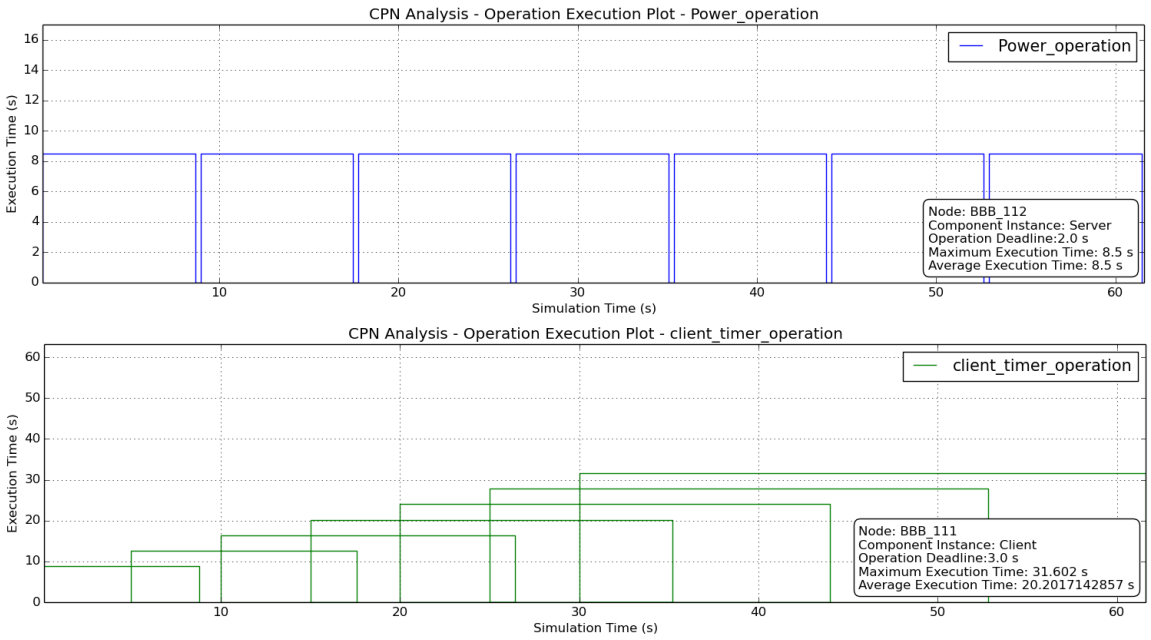

Figure 36: CPN Analysis Results: Client-Server Response Times in Bad Designs

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.

7.3.3 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.

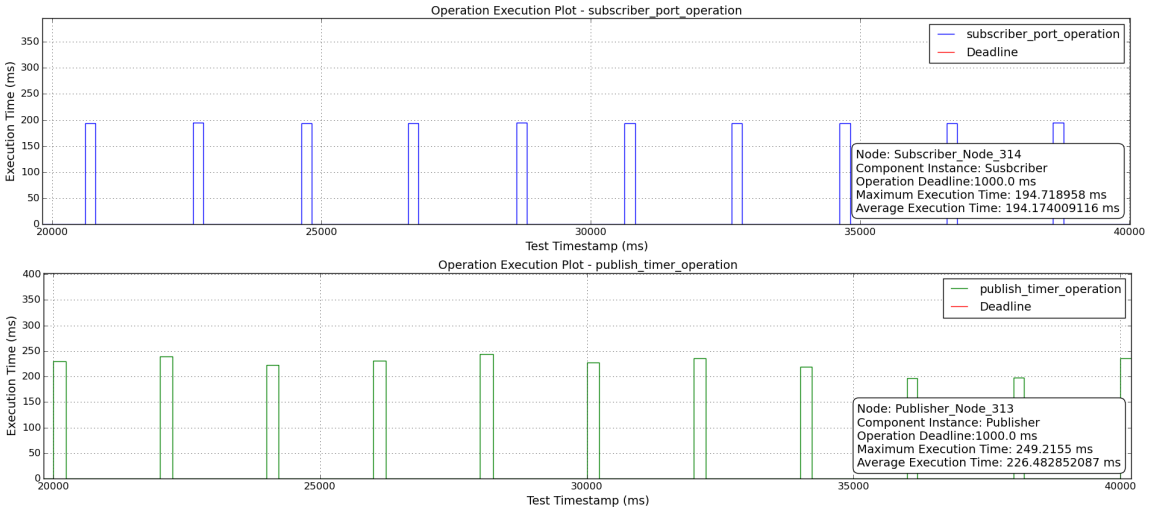

Figure 37: Experimental Observation: Publish-Subscribe Interactions

Figure 37 shows our testbed observations and Figure 38 shows our CPN analysis results. As evident, the CPN results closely match and validate this sample.

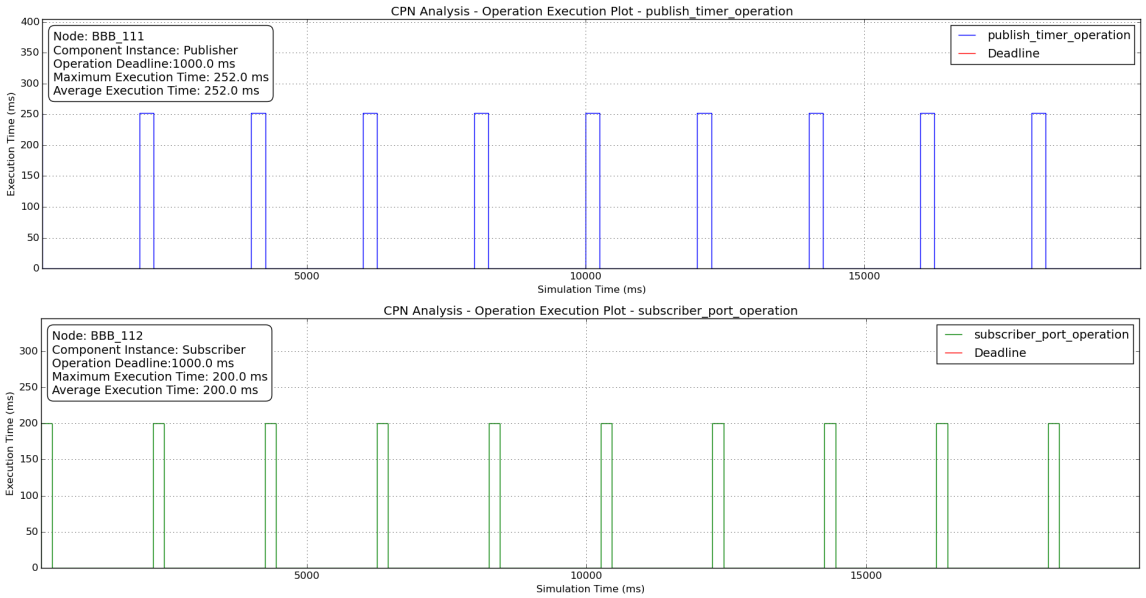

Figure 38: CPN Analysis Results: Publish-Subscribe Interactions

7.3.3.1 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 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.

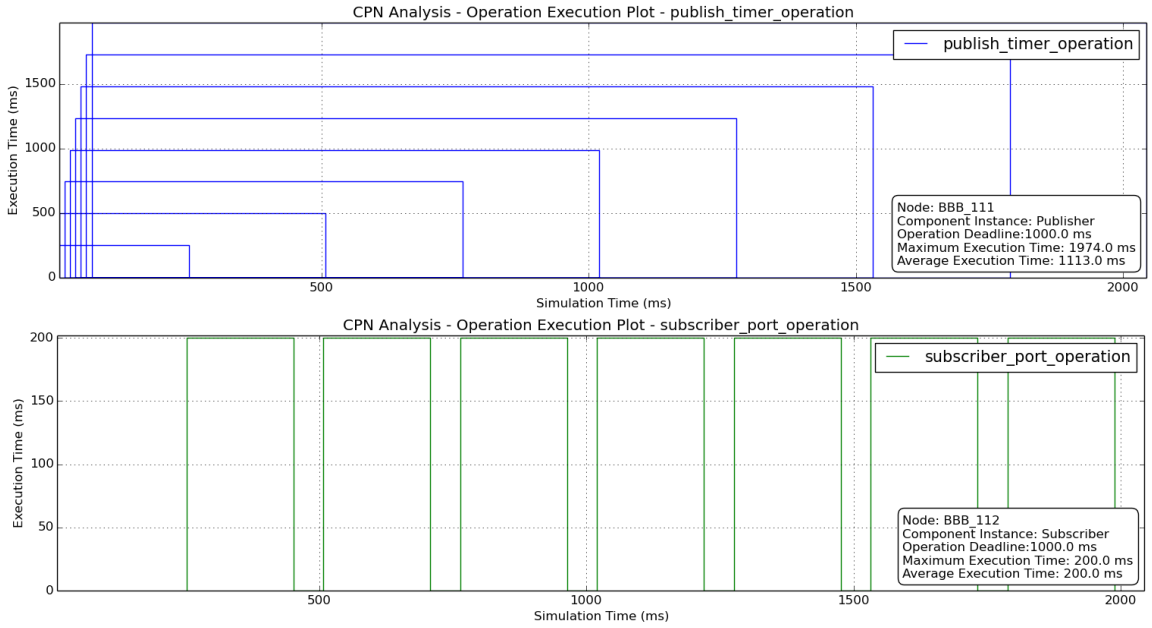

Figure 39: CPN Analysis Results: Time-triggered Publisher – Periodicity Issues

Figure 39 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.

7.3.4 Trajectory Planner

In the past [64], we have used a *Trajectory Planner* deployment to illustrate the utility of our state space analysis. Figure 40 shows the execution time plot of this sample. A Sensor component is periodically triggered every second by the *sensor_timer* at which point it

publishes a notification to the Trajectory Planner, 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.

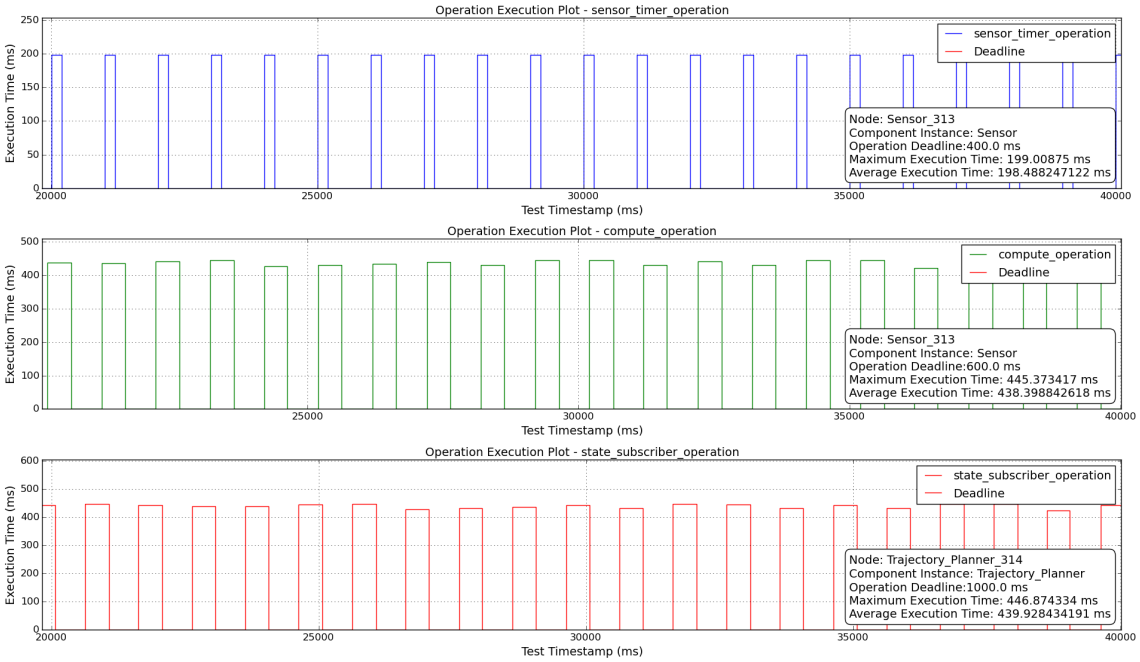

Figure 40: Experimental Observation: Trajectory Planner

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.

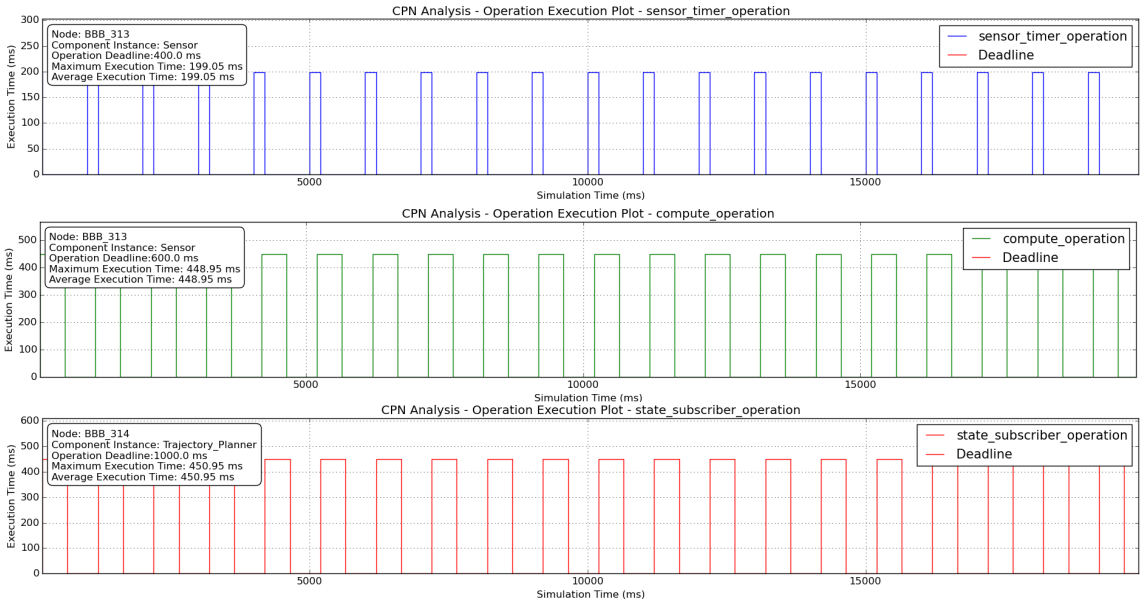

Figure 41: CPN Analysis Results: Trajectory Planner

7.3.5 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 42 shows our experimental observations on a 5-timer example.

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 43 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.

It must be noted here that the execution time values assigned to each timer operation in our CPN is the pure execution time i.e. the time taken for each timer operation to execute on the target CPU without interruption. This is the case for all operational execution times injected into the analysis model. If this is not done, then due to scheduling delays and interaction patterns, the CPN results will become gross overestimates.

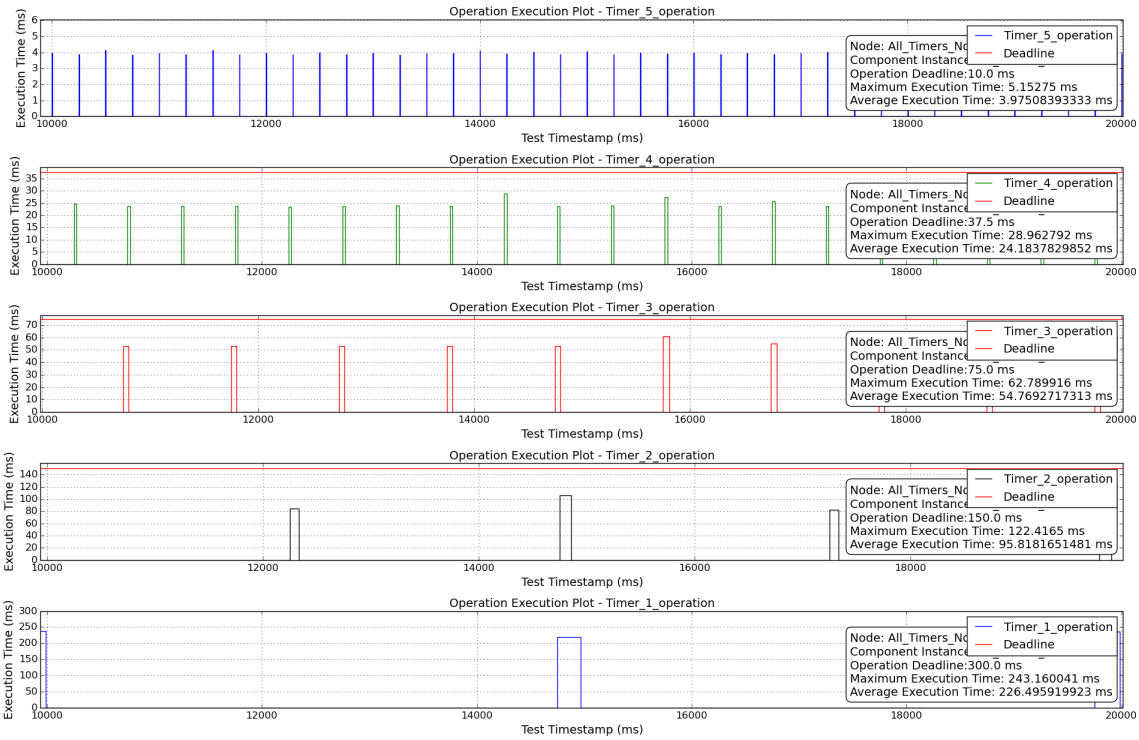

Figure 42: Experimental Observation: Periodic Timers

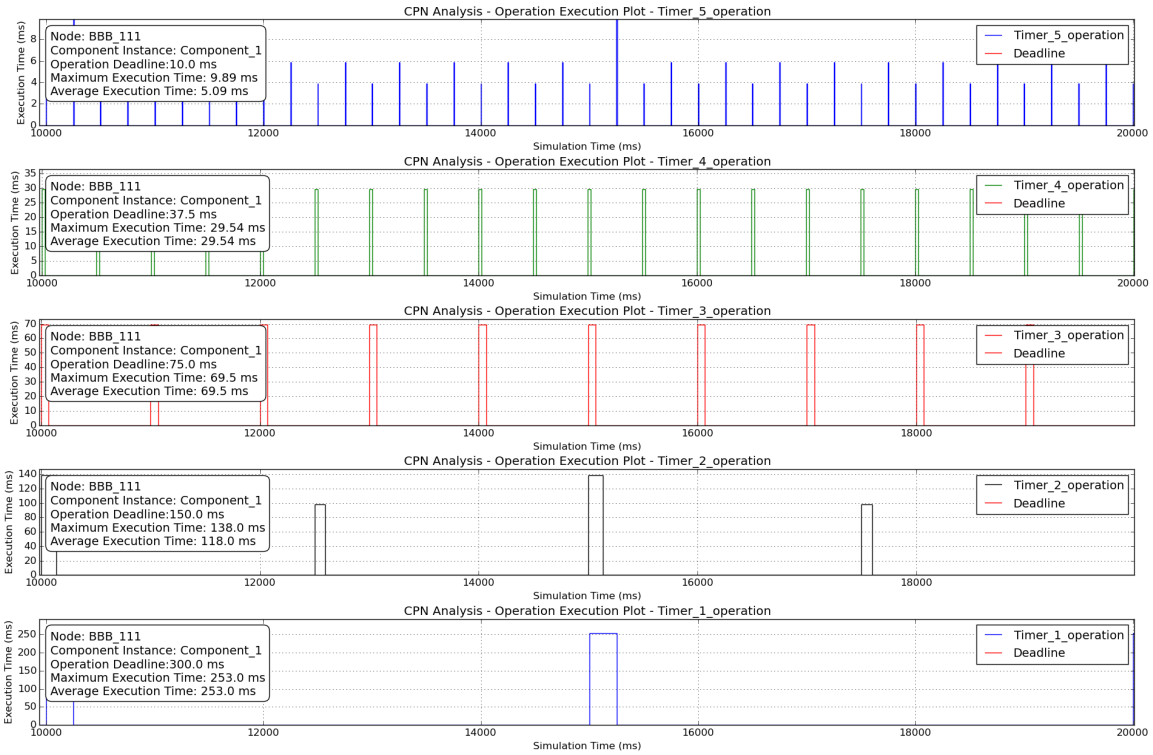

Figure 43: CPN Analysis Results: Periodic Timers

7.3.6 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 *cancellation* of long-running component operations to service other highly critical operations waiting in

the queue. With a few minor modifications to our scheduling schemes, long running 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.

7.3.6.1 Challenges

One of the primary challenges here is to identify the semantics of a long-running component operation i.e. the scenarios under which the component operations scheduler suspends a cooperating long-running operation in favor of some other operation waiting in the queue. If a long-running computation is modeled as a sequence of execution steps with bounded checkpoints, then the operation would execute one step at a time 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.

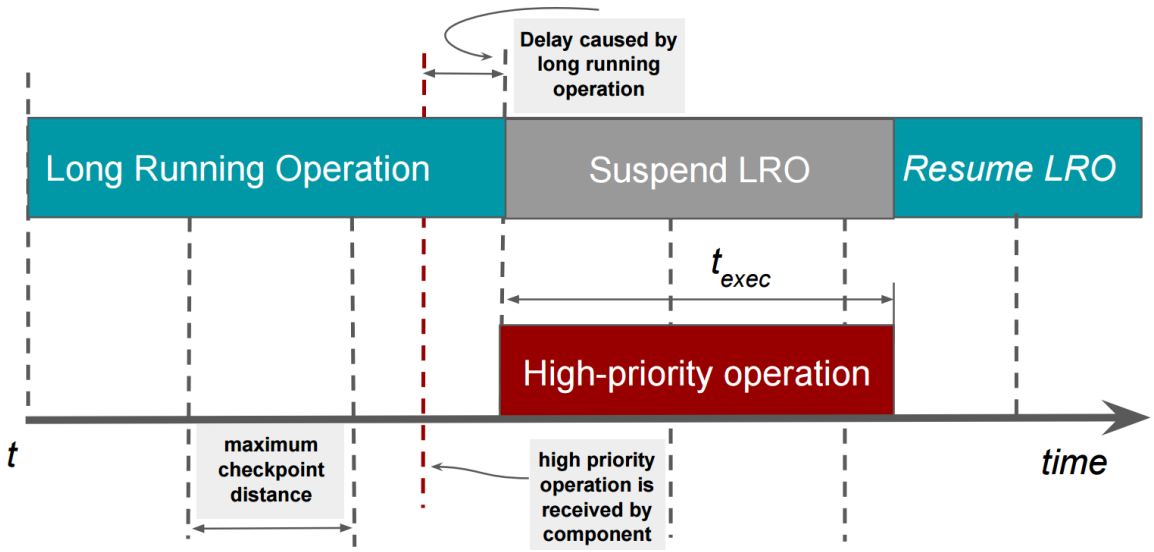

Figure 44: Long Running Operations - Timing Diagram

7.3.6.2 Implementation and Results

In each long-running operation, we, therefore, include a synchronous *checkpoint step*, as shown in Figure 44. 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. Figure 45 shows the *Software Model* for a component assembly with long running operations.

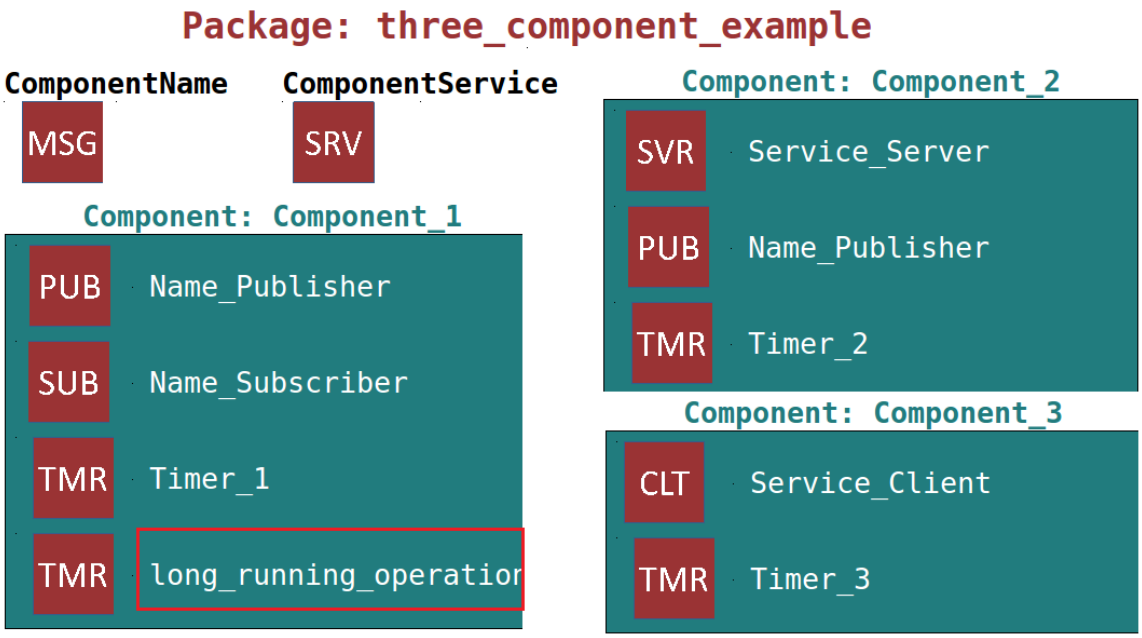

Figure 45: Long Running Operation - Software Model

The assembly consists of three components. Components *Component_1* and *Component_2* periodically publish on the *ComponentName* message. *Component_3* periodically queries the server in *Component_2*. During these interactions, *Component_1* is performing a long running operation, the duration of which, is magnitudes larger than the average execution time of all other operations. Figure 46 shows the execution time plot of this scenario, as measured on our testbed.

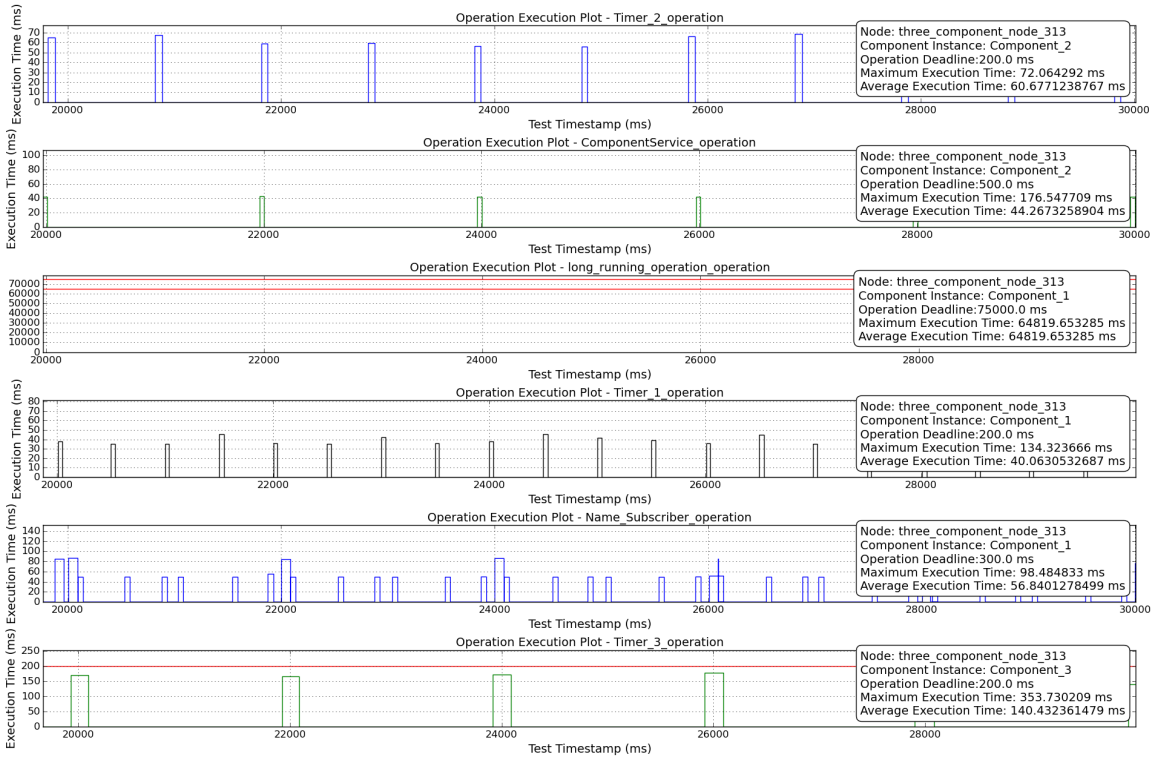

Figure 46: Experimental Observation: Composed Component Assembly

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 47 shows our CPN analysis results for the same assembly.

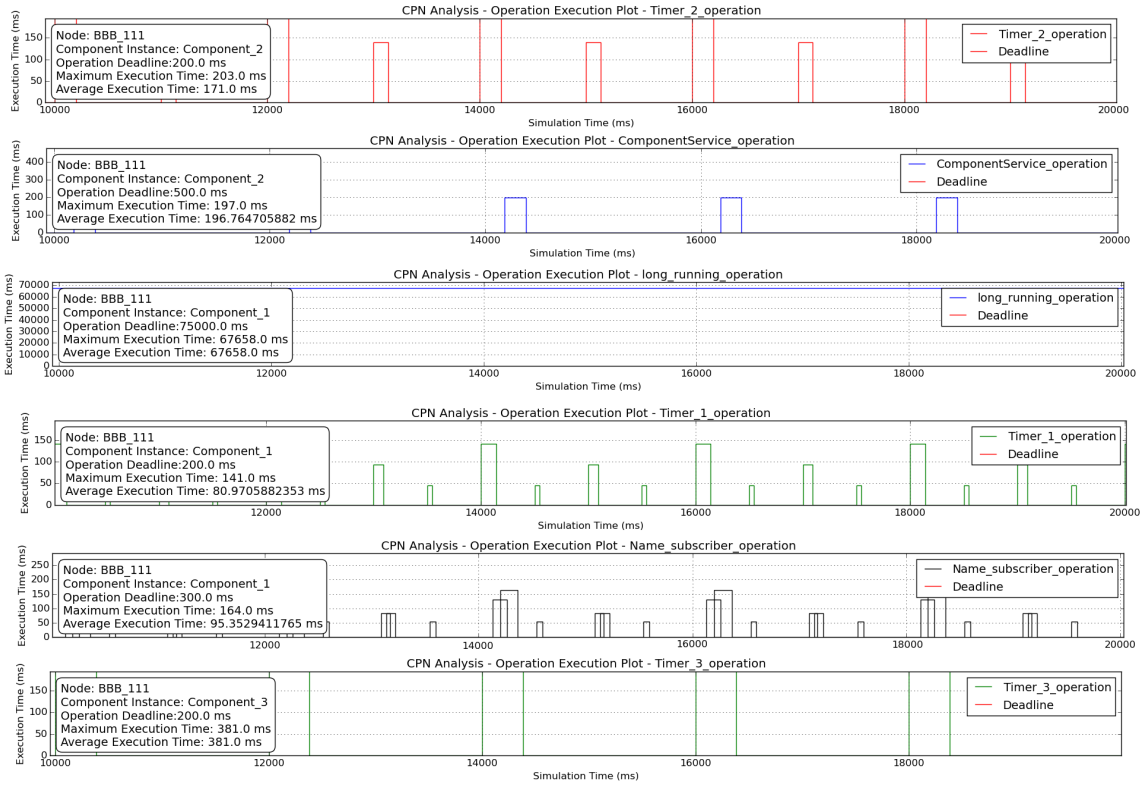

Figure 47: CPN Analysis Results: Composed Component Assembly

REFERENCES

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

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

[3] NASA CubeSats Mission to Mars. http://www.nasa.gov/press-release/nasa-prepares-for-first-interplanetary-cubesats-on-a

[4] Simulink. http://www.mathworks.com/products/simulink/.

[5] J. Abraham. Mathworks, natick, ma, 01760.

[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] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[8] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: A tool for schedulability analysis and code generation of real-time systems. In K. Larsen and P. Niebert, editors, *Formal Modeling and Analysis of Timed Systems*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer Berlin Heidelberg, 2004.

[9] D. P. Appenzeller and A. Kuehlmann. Formal verification of a powerpc microprocessor. In *Computer Design: VLSI in Computers and Processors, 1995. ICCD'95. Proceedings., 1995 IEEE International Conference on*, pages 79–84. IEEE, 1995.

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

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

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

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

[14] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. *UPPAAL-a tool suite for automatic verification of real-time systems*. Springer, 1996.

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

[16] N. S. Bjørner, Z. Manna, H. B. Sipma, and T. E. Uribe. Deductive verification of real-time systems using step. *Theoretical Computer Science*, 253(1):27–60, 2001.

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

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

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

[20] Y.-A. Chen, E. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao. Verification of all circuits in a floating-point unit using word-level model checking. In *Formal Methods in Computer-Aided Design*, pages 19–33. Springer, 1996.

[21] A. Cheng, S. Christensen, and K. H. Mortensen. Model checking coloured petri nets-exploiting strongly connected components. *DAIMI Report Series*, 26(519), 1997.

[22] S. Christensen, L. M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 450–464, London, UK, UK, 2001. Springer-Verlag.

[23] S. Christensen, L. M. Kristensen, and T. Mailund. *A sweep-line method for state space exploration*. Springer, 2001.

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

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

[26] A. David, J. Illum, K. G. Larsen, and A. Skou. Model-based framework for schedulability analysis using uppaal 4.1. *Model-based design for embedded systems*, 1(1):93–119, 2009.

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

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

[37] A. German. Software static code analysis lessons learned. *Crosstalk*, 16(11), 2003.

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

[39] 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.

[40] J. B. Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[41] M. J. Gordon and T. F. Melham. Introduction to hol a theorem proving environment for higher order logic. 1993.

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

[43] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[44] D. Henriksson, A. Cervin, and K.-E. Årzén. Truetime: Simulation of control loops under shared computer resources. In *Proceedings of the 15th IFAC World Congress on Automatic Control. Barcelona, Spain*, 2002.

[45] D. Henriksson, A. Cervin, and K.-E. Årzén. Truetime: Real-time control system simulation with matlab/simulink. In *Proceedings of the Nordic Matlab conference*, 2003.

[46] 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.

[47] G. J. Holzmann. An analysis of bitstate hashing. *Formal methods in system design*, 13(3):289–307, 1998.

[48] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

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

[50] K. Jensen. An introduction to the practical use of coloured petri nets. In *Lectures on Petri Nets II: Applications*, pages 237–292. Springer, 1998.

[51] K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[52] K. Jensen, L. M. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.

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

[54] S. C. Johnson. *Lint, a C program checker*. Citeseer, 1977.

[55] M. B. Jones. What really happened on mars, 1997.

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

[57] M. H. Kim and Y.-D. Kim. Simulation-based real-time scheduling in a flexible manufacturing system. *Journal of manufacturing Systems*, 13(2):85–93, 1994.

[58] 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.

[59] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *Model Checking Software*, pages 109–126. Springer, 2004.

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

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[66] 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.

[67] J.-L. Lions et al. Ariane 5 flight 501 failure, 1996.

[68] 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.

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

[70] P. Louridas. Static code analysis. *Software, IEEE*, 23(4):58–61, July 2006.

[71] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

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

[73] V. Massol and T. Husted. *Junit in action*. Manning Publications Co., 2003.

[74] 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.

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

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

[77] 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.

[78] J. Morrison and T. Nguyen. On-board software for the mars pathfinder microrover. In *Proceedings of the Second IAA International Conference on Low-Cost Planetary Missions*, 1996.

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

[80] Object Management Group. *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 edition, May 2003.

[81] 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.

[82] Object Management Group. *DDS for Lightweight CCM Version 1.0 Beta 2*. Object Management Group, OMG Document ptc/2009-10-25 edition, Oct. 2009.

[83] 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.

[84] W. R. Otte, A. Gokhale, D. C. Schmidt, and J. Willemsen. Infrastructure for Component-based DDS Application Development. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 53–62, New York, NY, USA, 2011. ACM.

[85] Y. Ouhammou, E. Grolleau, and J. Hugues. Mapping aadl models to a repository of multiple schedulability analysis techniques. In *Object/Component/Service-Oriented*

Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on, pages 1–8. IEEE, 2013.

[86] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In *Automated DeductionâĂŤCADE-11*, pages 748–752. Springer, 1992.

[87] 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.

[88] 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.

[89] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. 2007.

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

[91] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, 2000.

[92] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 134–143. IEEE, 1998.

[93] 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.

[94] 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.

[95] 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.

[96] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies.

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

[98] 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.

[99] 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.

[100] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

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

[102] G. Shelley and S. Forrest. Acl2 theorem prover.

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

[104] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: A flexible real time scheduling framework. In *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-time &Amp; Distributed Systems Using Ada and Related Technologies*, SIGAda '04, pages 1–8, New York, NY, USA, 2004. ACM.

[105] 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.

[106] J. R. Sklaroff. Redundancy management technique for space shuttle computers. *IBM Journal of Research and Development*, 20(1):20–28, 1976.

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

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

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

[110] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian. Configuring Real-time Aspects in Component Middleware. In *Proc. of the International Symposium on Distributed Objects and Applications (DOA)*, volume 3291, pages 1520–1537, Agia Napa, Cyprus, Oct. 2004. Springer-Verlag.

[111] 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.

[112] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275–298, 1998.

[113] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.

[114] 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.

[115] M. Westergaard and L. M. Kristensen. Two interfaces to the cpn tools simulator.

[116] 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.

[117] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):123–133, 1997.

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

[119] 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.

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

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

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