INTEGRATED TIMING ANALYSIS AND VERIFICATION OF COMPONENT-BASED DISTRIBUTED REAL-TIME SYSTEMS

By

Pranav Srinivas Kumar

Proposal (Area Paper)

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

SEPTEMBER, 2015

Nashville, Tennessee

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Safety and mission-critical distributed real-time embedded (DRE) systems are increasingly being used in a variety of domains such as avionics [17], locomotive control [112], and industrial control systems [113]. Given the dominant role of software in such systems, growing both in size and complexity, utilizing predictable and dependable software is critical for system safety. To mitigate this complexity, model-driven, component-based software engineering (CBSE) and development [13, 21, 38] has become an accepted practice. CBSE tackles the increased demands with respect to requirements engineering, high-level design, design error detection, tool integration, verification and maintenance. The widespread use of component technology 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. Component models describe the software components that are used to build the system. The reusable nature of components leads to developers focusing on other critical parts of the system design, leading to shorter development cycles and reduced costs.

Complex, managed systems, e.g. a fractionated spacecraft following a mission timeline and hosting distributed software applications expose heterogeneous concerns such as 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 failure recovery tasks that are periodically undertaken on the distributed nodes. Once deployed, it is often difficult to obtain low-level access to such remote systems for run-time debugging and evaluation.

These types of systems therefore demand advanced design-time modeling and analysis methods to detect possible anomalies in system behavior, such as unacceptable response times, before deployment.

Our team has designed and prototyped a full information architecture called **D**istributed **RE**al-time **M**anaged **S**ystem (DREMS) [25, 28] that addresses requirements for rapid component-based application development. In prior work, we have described the design-time modeling capability [26], and the component model used to build and execute applications [77]. The formal modeling and analysis method presented in this paper 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 1 shows a *spiral model* [15] of a typical industrial software/system development life cycle. The five primary stages in this loop are: (1) Requirements analysis, (2) software design, (3) implementation, (4) integration (5) testing and debugging, (6) design evolution. Any system design first begins with an informal description of the requirements. Analyzing the requirements yields the necessary layers of software (or hardware) required in the system design. Once the design is complete, the pieces/layers of software are implemented by software developers. These pieces of software i.e. components are then *integrated* together to obtain the overall system. A separate team of testers check the integrated system using various testing methods including unit tests and integration tests. Once the software has met certain testing requirements, the software becomes more robust. Inconsistencies identified during this testing process often causes changes to high-level requirements. This

cycle repeats as long as the software is *alive*. Any new features to be integrated in this system need to go through these steps.

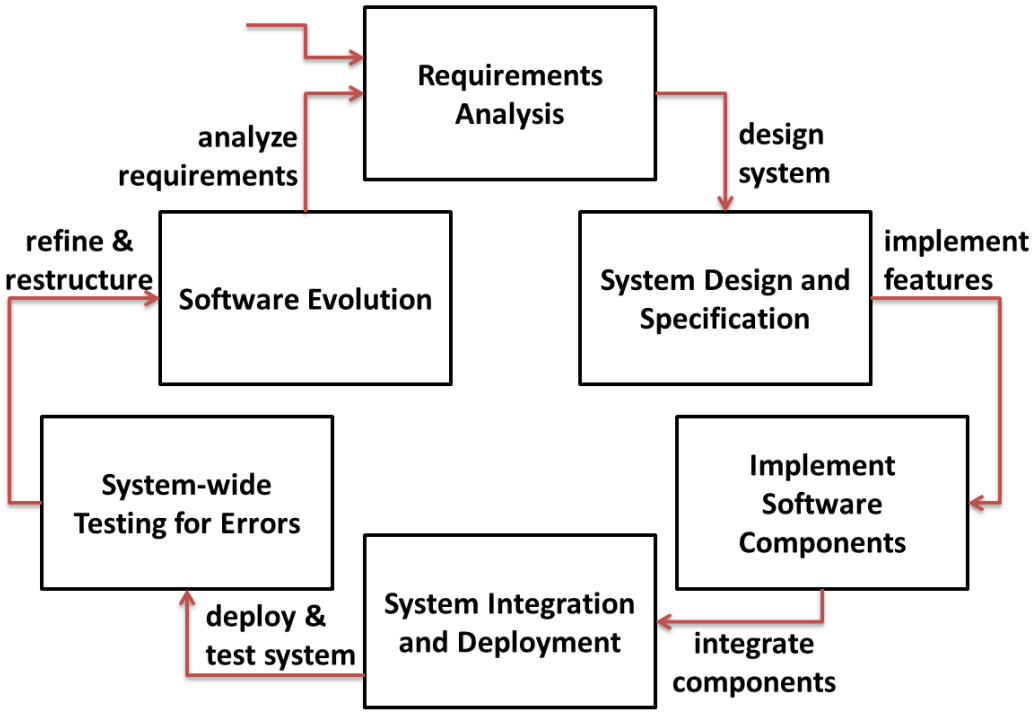

Figure 1: Industrial Software/System Development Life Cycle

The analysis work proposed in this thesis supports a verification-driven workflow for component-based software development, as shown in Figure 2. Application developers use domain-specific modeling languages to model the component assembly, interaction patterns, component execution code, sequence of operations, and associated temporal properties such as estimated execution times, deadlines etc. Using such application-specific parameters in the *design* model, a Colored Petri net-based (CPN) [47] *formal analysis model* is generated. The system behavior is both simulated and analyzed using a CPN execution engine, CPN Tools [88], 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.

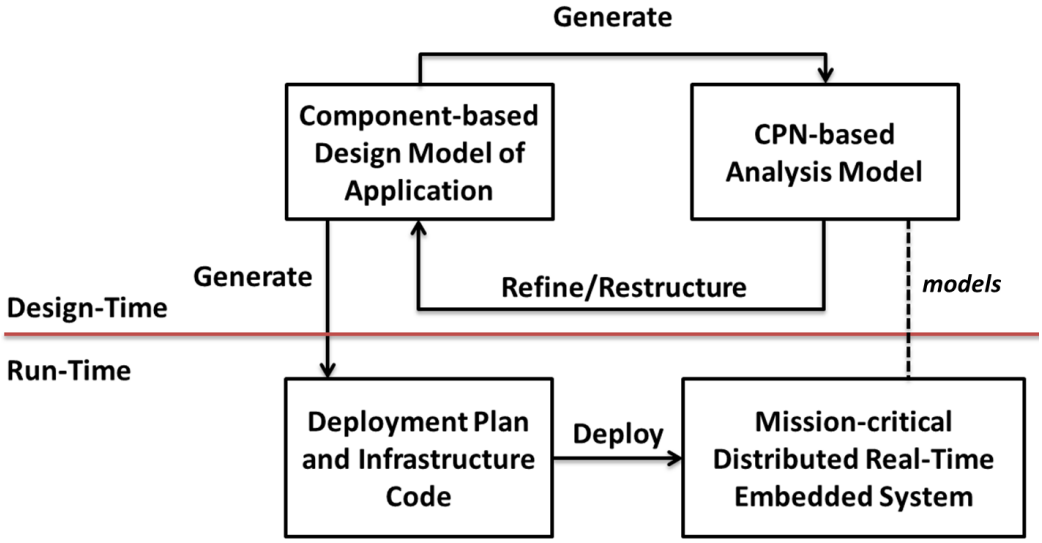

Figure 2: Verification-driven Workflow

The results of this analysis help improve the structure of the application, enabling safe deployment of dependable components that are known to operate within system specifications. Some implicit assumptions to this analysis include a prior knowledge of WCET estimates for the various code blocks in the component execution code. When designing the overall 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 high-level requirements and appropriately distributed between the different components in the system. The analyzed system may not necessary be complete, but instead be in a process of evolution. As the design progresses, the system requirements become extended and the design is re-verified at each stage to ensure the timing guarantees advertised.

The remainder of this proposal 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 describes the related research in timing analysis and verification for distributed real-time embedded applications. Chapter IV introduces the DREMS infrastructure and Component Model used to experiment with and validate the timing analysis results. This chapter also proposes the Colored Petri net-based timing analysis methodology devised for component-based DRE systems, including published results and proposed contributions. Chapter V concludes the proposal, providing a tentative timetable for the successful completion of the proposed work. Finally, Chapter VI lists relevant publications.

CHAPTER II

FUNDAMENTALS

A real-time system [64] 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) [109] or response times [52]. 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*.

There are several challenges to modeling and analysis of such distributed safety-critical systems. Using real-time components that can run on heterogeneous hardware platforms means that the components have different timing characteristics on the different platforms. Therefore, a component must be molded and re-verified for each hardware platform on which it is deployed. Secondly, component-based systems are constructed by assembling a tested set of components. Two components that individually provide timing guarantees may not aid the overall system-level requirements when executed concurrently in a specific hardware platform. The challenge here is ensuring that a system consisting of composed set of verified components still retains its timing behavior. Thus, the requirements for timing analysis become two fold:

- Verify the timing properties of each component in the system - Does the operational behavior of a software component meet its timing requirements?
- Analyze the schedulability of a component assembly - When composed together, do all components work as expected to meet the end-to-end system-wide timing requirements?

The results of this two-level timing analysis should indicate with sufficient proof the stability or instability of the composed component-based system. Achieving this workflow is the fundamental goal of our timing analysis methodology, presented in later sections. Our

analysis model uses a Colored Petri Net [47] based formal analysis methodology. However, motivating this methodology requires first an in-depth literature review. The following section reviews both general analysis methodologies used in the past, and specific system-level modeling and timing analysis techniques for concurrent real-time systems, along with advantages, limitations and potential improvements that motivate our proposed work.

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 [36] tests in code evaluate performance in the presence of bad input data. Redundant hardware measures e.g. processors in the Space Shuttle [99] 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 [55], 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 [84]. 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 [39], execution time of a piece of software becomes non-deterministic [89].

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 [106] and other genetic algorithms [85, 107] 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 [41, 53]. The TRUETIME simulation toolbox [40], based on MATLAB/Simulink [96], 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 [66] 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 [66] 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 [62] and the Mars Pathfinder [51, 72].

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 [8, 18]. 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 [44], UPPAAL [61] and Kronos [110].

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 [14], 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) [32]. 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 [37], and the Prototype Verification System (PVS) [80]. 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 [80] requires the user to suggest case splits, lemmas, and so on during the course of the proof. A non-interactive prover such as ACL2 [95] 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 [50], 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 [68] in Java is a useful tool for writing tests. Static code analysis [65] 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 [4] 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 [33] :

- 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 [33].

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 [97] 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 [63]: 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 [12, 22, 61] 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 [6], 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 [7] 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 [83] 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 3 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 3, 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

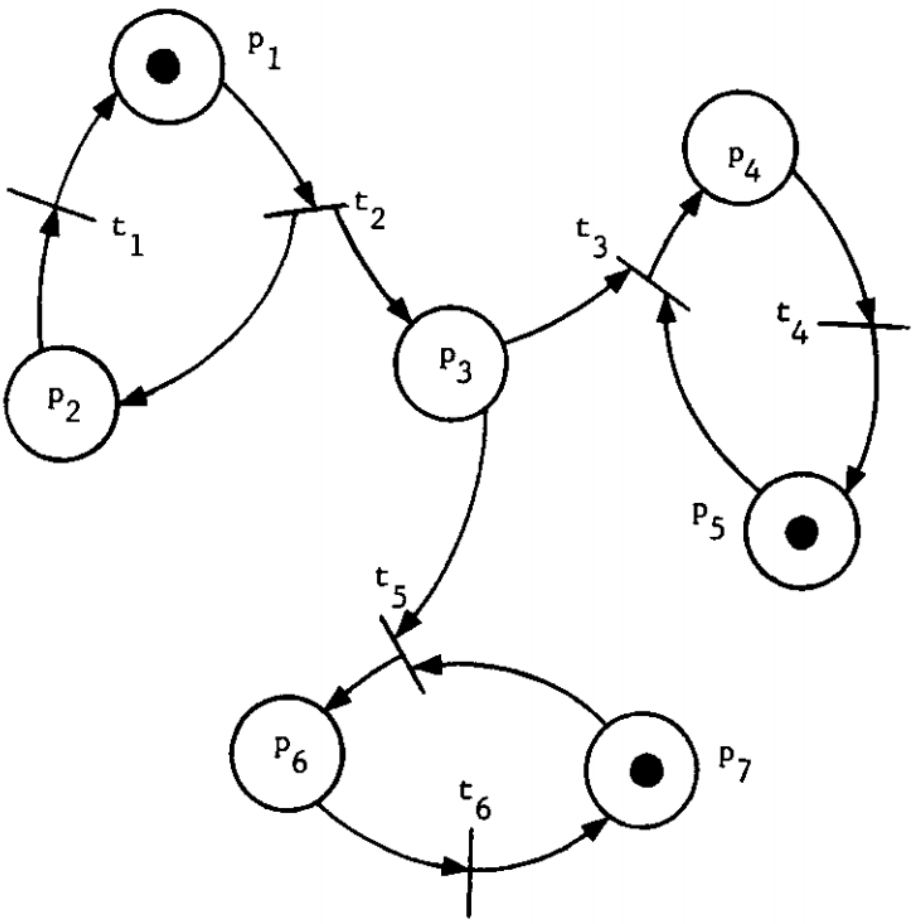

Figure 3: Sample Petri Net, reprinted from [83]

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 3, 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 [23, 42], 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 [42] and timed systems [114]. For a detailed study of Petri nets and its applications, the reader is referred to standard textbooks [83, 90] and survey papers [73, 111, 115].

Petri nets have evolved through several generations from low-level Petri nets for control systems [90] to high-level Petri nets for modeling dynamic systems [49] to hierarchical and object-oriented Petri net structures [24] that support class hierarchies and subnet reuse. Several extensions to Petri nets exist, depending on the system model and the relevant properties being studied e.g. Timed Petri nets [103], Stochastic Petri nets [11, 67] 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 [88] 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 [92]. Development tools like MARTE [75] and AADL [30] 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 [92], the authors propose a bridge that translates AADL specifications of real-time systems to Petri nets for timing analysis. This formal notation is deemed to be well-suited to describe and analyze concurrent systems and provides a strong foundation for formal analysis [34] methods such as structural analysis and model checking. The high-level goal is to check and verify AADL models for properties like deadlock-freedom and boundedness. The workflow presented here is similar to the 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 [98]. 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 [92], have also investigated AADL model analysis using Petri net extensions such as Timed Petri nets [91]. 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 [5] that observe the system behavior for property violations and prompt completion of operations. However, this work [91] 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 [79] 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 [35] 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 [54, 63] and distributed real-time systems [82, 100].

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 [71].

For analysis, the MAST suite includes schedulability analysis tools that use newly published research techniques such as the offset-based methods [81] that enhance analysis

results, providing less pessimistic estimates than previous results [100]. 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 [94] 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 [16]. 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 [31] 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 [32] 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 [69, 70] 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

PROPOSED WORK: DESIGN-TIME TIMING ANALYSIS OF COMPONENT-BASED DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS

4.1 Colored Petri net-based Modeling and Analysis Methodology

4.1.1 Design Model: Distributed Managed Systems (DREMS)

The target component model and architecture used to illustrate the proposed timing analysis methods is the DREMS infrastructure (**D**istributed **RE**altime **M**anaged **S**ystem) [28], [77]. DREMS was designed and implemented for a class of distributed real-time embedded systems that are remotely deployed and are characterized by strict timing requirements e.g. a cluster of satellites. 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 [26] that integrate with a well-defined and fully implemented component model [60, 77] 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.

The DREMS component model is based on the Component Integrated ACE ORB (CIAO) [104, 105] project. CIAO is an implementation of the OMG's Lightweight CORBA Component Model (CCM) [74]. CIAO uses the TAO [93] CORBA object request broker (ORB) as its default underlying communication middleware. With the recent standardization of connector mechanisms [76], 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 [78] enabling a variety of interaction

schemes. For safe and deadlock-free behavior, this component model also allows only one thread of control per component to be active at any given instant of time.

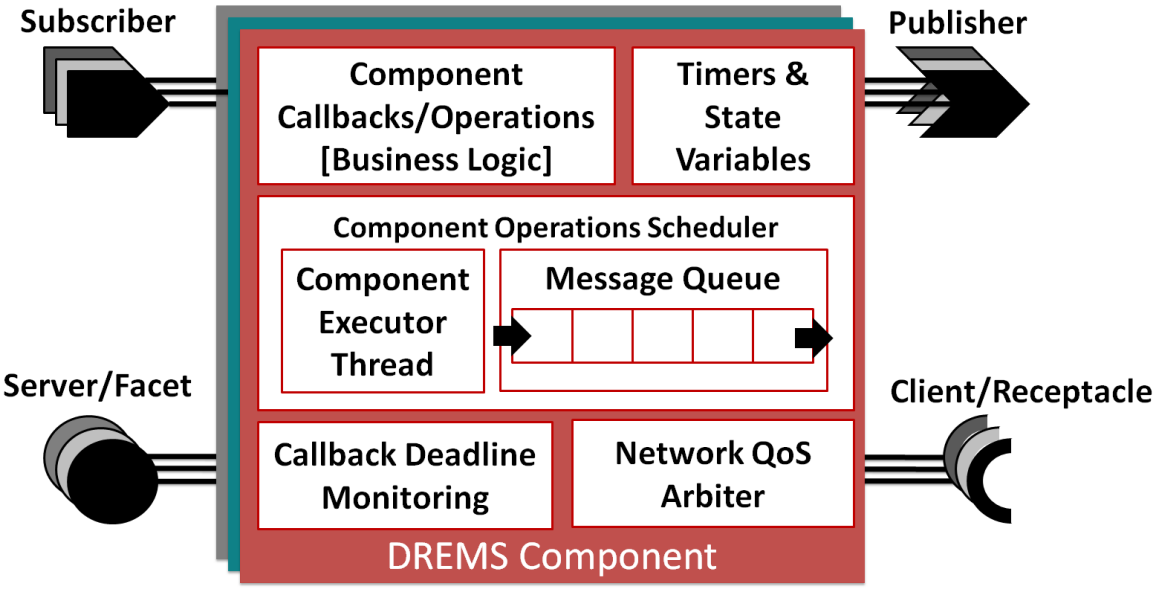

Figure 4: DREMS Component

Figure 4 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) [87, 102]. 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 [29] style of messaging. More details on this component model can be found in [77].

4.1.1.1 Component Operations

An operation is an abstraction for the different tasks undertaken by a component. These tasks are implemented by the component's executor code written by the developer. Application developers provide the functional, *business-logic* code that implements operations on the state variables e.g. a PID 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 instances of operations ('messages') that are ready for execution and need to be serviced by the component. These operations service either interaction requests (seen on communication ports) or service requests (from the underlying framework). An example for the latter is the use of component timers that can periodically (or sporadically) activate an operation.

Figure 5 shows the basic structure of this model. Each operation is characterized by a priority and a deadline. Operation deadlines are quantified in absolute time measured starting from when the operation is enqueued onto the component message queue. These operations are sorted and scheduled based on one of three scheduling schemes: Earliest Deadline First (EDF), First In First Out (FIFO), or Priority FIFO (PFIFO).

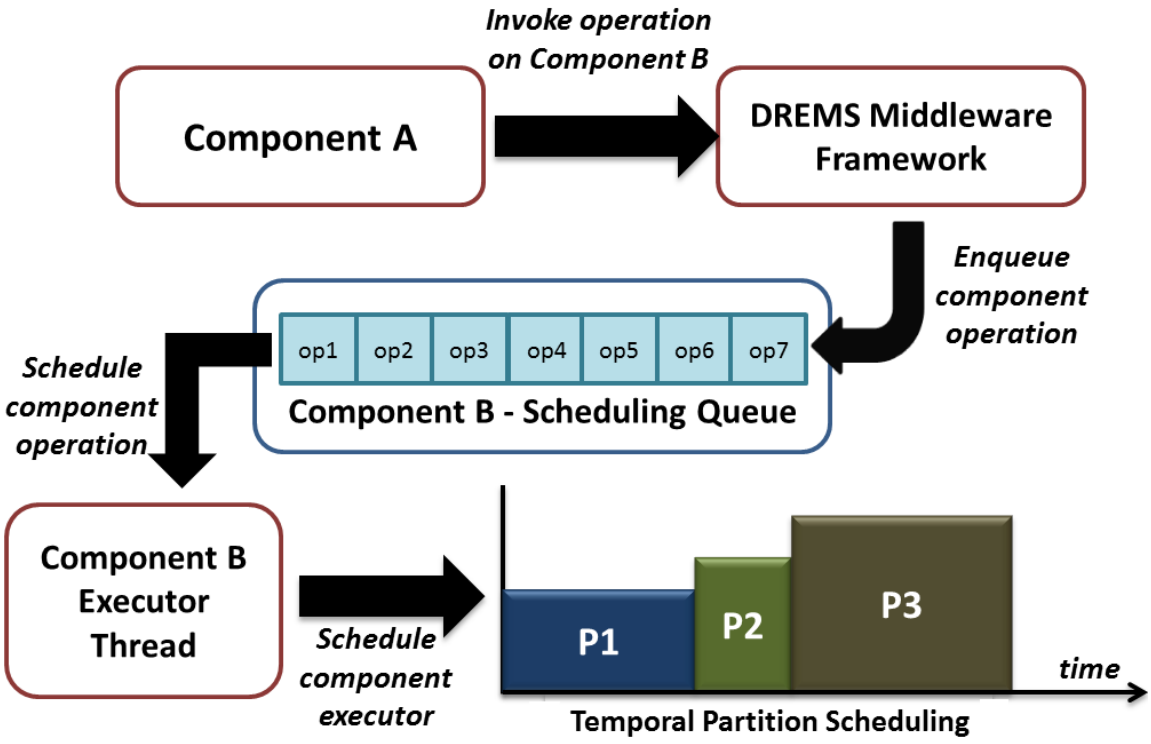

Figure 5: Scheduling Component Operations

To facilitate component behavior that is free of deadlocks and race conditions, the component's execution is handled on a single thread. Operations in the message queue are therefore scheduled one at a time under a non-preemptive policy. A component dispatcher thread dequeues the next ready operation from the component message queue. This operation is scheduled for execution on a component executor thread. The operation is run to completion before another operation from the queue is serviced. This single-threaded execution helps avoid synchronization primitives such as internal state variables that lead to strenuous code development. Though components that share a processor still run concurrently, each component operation is executed on a single component-specific executor thread.

Figure 6 shows a sample timing diagram of how a component operation is handled. At

time 0, the component executor thread is running some previously ready component operation, *op1*. At this point, consider that a new component operation *op2* is enqueued onto the message queue and marked as ready. At time 10, assuming that no other component requires scheduling, the component dispatcher thread of this component dequeues the next ready operation for execution. Assuming the component executor thread is scheduled immediately by the underlying processor, this thread runs the ready operation by invoking its *execute* function. This operation is run to completion at time 16. The total time taken for execution of this operation is measured from when the operation was enqueued, i.e. time 0. If the time taken for the operation exceeds its deadline, a fault manager is immediately notified. The duration of the component operation is further delayed by temporal partitioning enforced by the OS scheduler. This adds to the need for schedulability analysis, especially in case of safety and mission-critical applications.

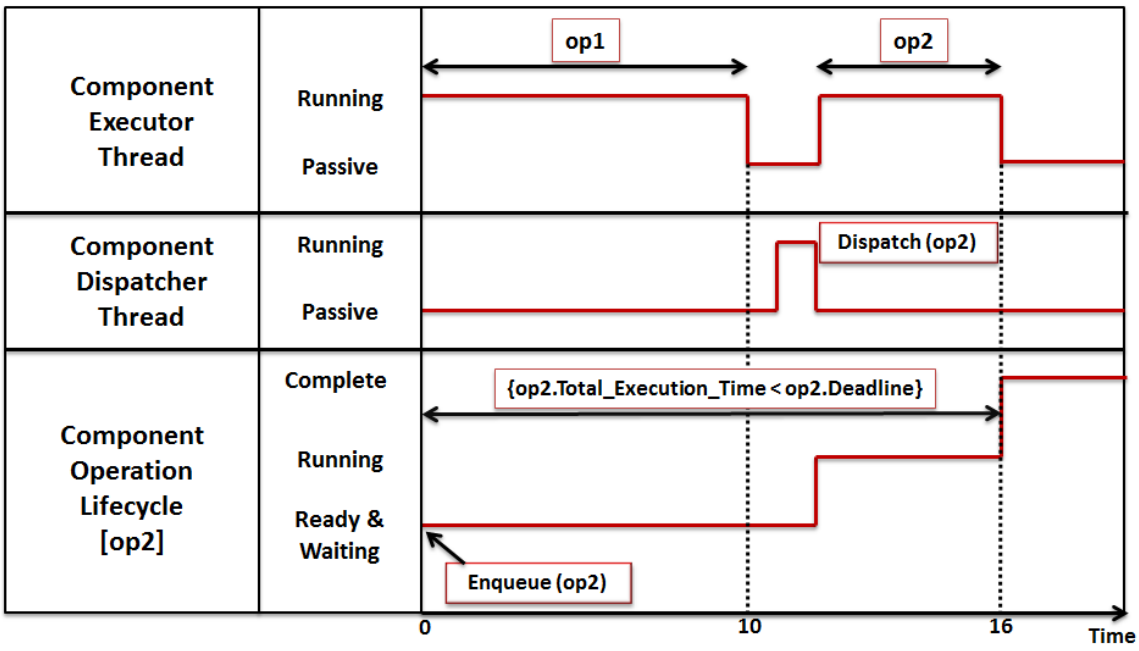

Figure 6: Component Operation Execution

4.1.1.2 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 [9] style temporal and spatial partitioning scheme.

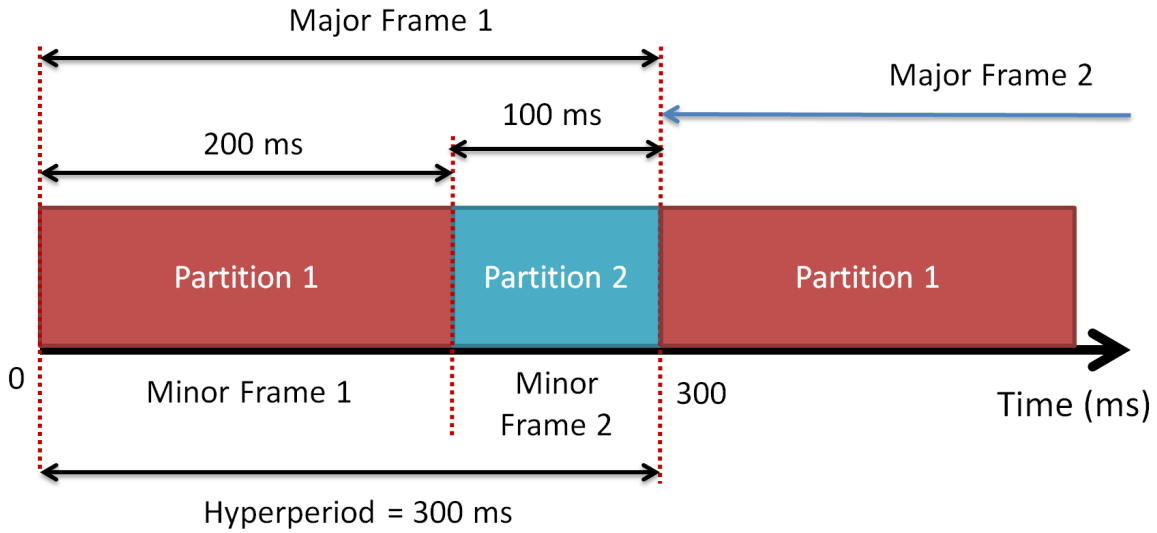

Figure 7: Sample Temporal Partition Schedule with Hyperperiod = 300 ms

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 7 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. Although the proposed analysis work tackles the challenges of a hierarchical scheduling scheme such as in DREMS, the presented analysis also studies cases without temporal partitioning, relying on the default Linux scheduling scheme.

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 that the developer specifies. 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 [105] and ACM [27], and the execution semantics are precisely defined and implemented. A qualitative evaluation of its capabilities [77] show that although the model was designed for fractionated spacecraft, DREMS is suitable for a variety of distributed and embedded environments.

4.1.2 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 include:

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.

4.1.3 Challenges

4.1.3.1 Modeling Component-based Applications

The CPN model should capture the behavioral semantics of our component model described in [77], using knowledge of several factors that resolve the deployment of the component-based application. These factors include the following system properties: (1) configuration of temporal partition scheduling on each node of the distributed system, (2) location of each component being deployed (which temporal partition and which computing node) (3) properties of the component executor threads (thread priority), (4) properties of timers (period and offset), and (5) component interactions and assembly (i.e. the 'wiring'). The timing and behavior of the component interactions is dependent on both the application developer's business logic code and the distributed *deployment plan*. Poorly written application code could cause circular dependencies, deadlocks or ever-growing message queues that are not easy to identify from a high-level system design model. The goal of the CPN model is to establish a simulation medium and an analysis framework that can identify and alert designers about such design inconsistencies.

4.1.3.2 Analysis Challenges

Our CPN-based analysis methodology is a type of formal verification technique. There are several challenges to analyzing large DRE systems using formal verification methods like this. For an executable system model, a bounded state space needs to be generated. Here, the state space represents a tree of possible behaviors that the system can exhibit when executing from a known initial state. The state space should comprise of every possible combination of execution traces taken by system state variables. Once the evolution of state variables is recorded, state space analysis becomes a graph searching problem where given a state space graph of finite size i.e. fixed number of nodes and edges, how efficiently can the graph be searched to identify breakpoints or interesting graph nodes that represent

positive or negative system states. The graph nodes each contain information about the system state variables e.g. execution time, thread state etc. By querying each node in the state space, system properties such as deadlock-freedom, deadline violations and trigger to response times can be deduced. Challenges to this problem include (1) state space explosion when the number of variables in the system grows, (2) efficiency of the searching algorithm e.g. memory consumption, worst-case search time etc., and (3) size of the bounded state space i.e. is the generated state space a complete representation of the system execution. The bound on the state space determines the validity of the state space analysis; the generated state space should span a much longer time compared to the periodicity of the DRE system being analyzed. The size of the generated state space is dependent on the amount of concurrency in the behavior e.g. if all the executing threads have unique priorities, the thread execution order is a constant as the scheduling is priority-based. However, for large systems with groups of applications and increased concurrency, an equally large state space is required to observe the tree of possible thread executions and operational behaviors.

From an analysis perspective, there are some important considerations that need to be made when building this CPN model. Foremost among these, is the issue of scalability. Here, a scalable analysis model needs to be efficient in two ways: (1) How does the CPN model grow in size with changes/additions to the system design model? Constructing/Generating a new CPN *place* for each new added component port or hardware computer is not efficient because the CPN model would not scale well for large systems with hundreds of components; (2) How does the state space generated by the CPN model scale? How is the time taken to traverse the system state space affected when the analysis model has to handle hundreds of threads instead of tens of threads? These challenges need to be addressed for the analysis model to be useful in real-world DRE system scenarios e.g. airplane control systems and vehicle ECU networks.

4.1.3.3 Generating CPN Analysis Model from System Design Model

For large distributed applications, with multiple timers and component interaction patterns, hand-writing the CPN token specifications will prove to be both cumbersome and error prone. To avoid this, the temporal behavior specification discussed in Section 4.2 for component operations needs to be integrated into the system design model. Any part of a component e.g. timers, server ports, subscriber ports etc. that exposes an operation to the external environment needs to be *tagged* with an abstract business logic model. This model describes the sequence of steps executed each time the operation is scheduled. Such models contain vital information that drives the CPN analysis execution. The business logic models, along with temporal partitioning specifications, component definitions and assembly, and the deployment plan need to be parsed, interpreted and translated. The system specifications derived from these sub-models need to be converted into Colored Petri net tokens that are injected into various parts of our generic CPN model.

4.1.4 The Choice of Colored Petri Nets

With Colored Petri Nets (CPN) [47], tokens contain values of specific data types called colors. Transitions in CPN are enabled for firing only when valid colored tokens are present in all of the typed input places, and valid arc bindings are realized to produce the necessary colored tokens on output places. The firing of transitions in CPN can check for and modify the data values of these colored tokens. Furthermore, large and complex models can be constructed by composing smaller sub-models as CPN allows for hierarchical description.

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.

4.1.5 Outline of Solution

The top-level CPN Model is shown in Figure 8. The places (shown as ovals) in this model maintain colored (typed) tokens that represent the states of interest for analysis e.g. the *Clocks* place holds tokens of type *clock_token* maintaining information regarding the state of the clock values and temporal partition schedule on all computing nodes.

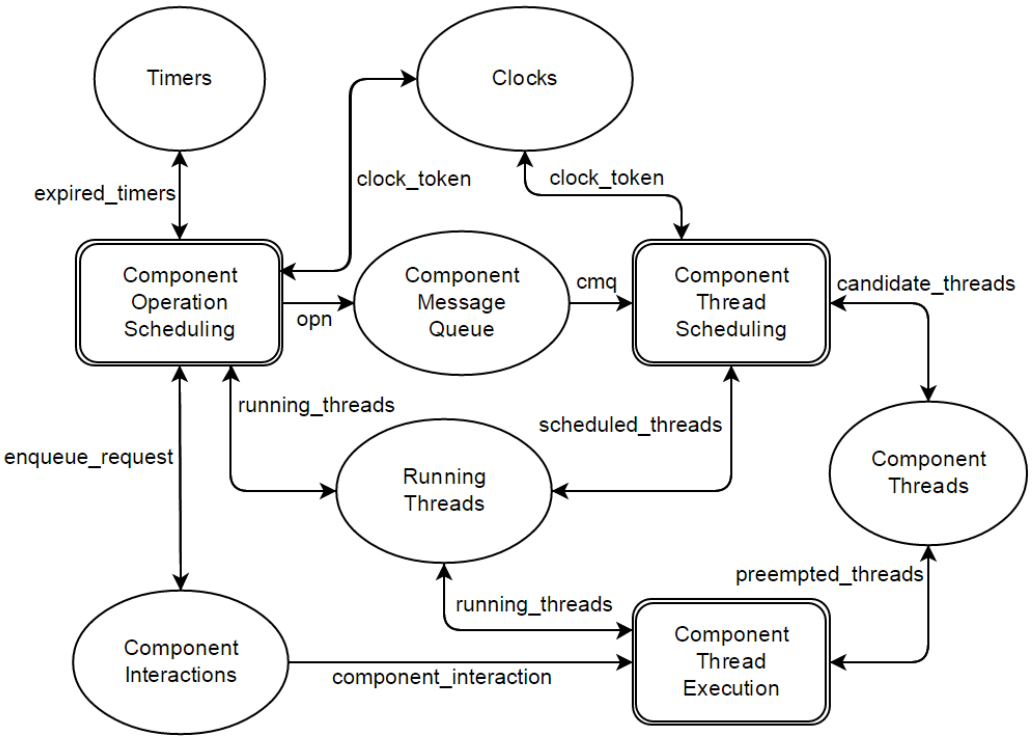

Figure 8: Hierarchical Colored Petri Net Analysis Model

The above model contains three hierarchical transitions that abstract low-level modeling details. These transitions handle component operation scheduling, OS thread scheduling,

and execution, each progressing the hardware clocks appropriately. Transitions in Colored Petri nets can have *priorities* i.e. a high-priority transition may be preferred over other enabled transitions in the marking. Although the low-level details of this model are not within the scope of this report, we believe that this model is effective in representing the complexities of component-based distributed real-time applications. Using a 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 etc.

4.1.6 Evaluation of Solution

Rigorous evaluation of our solution is an important requirement for both justifying our approach and identifying its advantages and disadvantages. Evaluating our CPN-based analysis model requires us to probe both it's functional and behavioral aspects. The CPN model must accurately represent both the structural aspects of a software deployment e.g. process-to-hardware mapping, component operational steps etc. and the behavioral aspects e.g. thread scheduling, blocking effects, execution delays etc. This can be done by formally representing our DREMS component execution semantics and comparing this representation with our CPN formal model. Execution traces generated by CPN simulations for a variety of example cases would show the system elements modeled appropriately e.g. component operation scheduling, time-triggered periodic execution etc. Simulating both single-hardware and multi-hardware deployments presents us with trace data that can be compared with real-world deployments. Such comparisons, if equivalent, provide us the confidence required when selecting such an analysis model as an appropriate one. Similarly, the state space analysis can be evaluated by (1) forcing negative results e.g. deadlocks, deadline violations in the real system, then (2) modeling this system using our CPN analysis model, and (3) ensuring that the state space analysis process catches the timing violations. Since the presence of a timing violation in the CPN analysis does not necessitate a timing anomaly in the real-system, the opposite scenario is forced for the purpose of

evaluation. Lastly, with peer-review evaluation of the structural and behavioral CPN model of the DREMS system, the design-model to CPN model translation is implicitly evaluated too. By forcing the use of the *CPN Analysis Model Generator*, we evaluate the translation rules for a wide range of examples.

4.1.7 Contributions

4.1.7.1 Colored Petri Net-based Model

We designed and built a Colored Petri net-based extensible and scalable analysis model of distributed component-based applications. Several CP-nets are used to compose the different layers of the design model e.g. hierarchical scheduling, network-level interactions etc. This modeling paradigm was reported [60] with a simple *Trajetory Planner* example - a distributed real-time scenario with temporal partitioning and a combination of interaction patterns supported by DREMS, including synchronous remote procedure calls and anonymous publish/subscribe message passing schemes. Improvements to this model have since been made and are discussed in later sections.

4.1.7.2 State Space Analysis and Verification of Safety-critical System Properties

We used the prototype CPN analysis model and performed bounded state space analysis using both the in-built state space analysis tool in CPN Tools [88] and also user-defined application-specific queries e.g. deadline violation checks, partial thread execution order generation etc. Here, tokens in CPN places, in each marking, encode the state of the system at that marking (time instant). State space analysis yielded positive results [60] identifying deadline violations, deadlocks, worst-case trigger-to-response times, and partial thread execution orders in a variety of scenarios. Scalability testing proved the CPN model to be effective for medium-large sized component-based applications - tested with up to 100 components distributed on up to 10 computing nodes.

4.1.7.3 Design Model to Analysis Model Translation

A *CPN Generator* model interpretor [60] was developed to generated Colored Petri net analysis models from system designs. This interpreter parses the design model tree, identifies system properties such as component definitions, port properties, scheduling schemes etc. and constructs an interpreted data structure (tree). Using this tree, system properties are converted to relevant CPN token strings. Using T4 Text Templates [101] in a Visual Studio environment, the generated strings were embedded into a generic CPN text template by a templating engine. The end result is a parameterized Colored Petri net (.cpn) file that can analyze the system. Since then, we have also ported this generator to a Python-based development environment [58] and performed further testing.

4.2 Modeling Component Operation Business Logic

4.2.1 Problem Statement

Consider a set of component-based applications deployed on distributed hardware. Each applications consists of groups of components that interact with each other and also with the external environment e.g. I/O devices, other applications, underlying middleware etc. Each component exposes a set of interfaces through which external entities can request *operations*. As mentioned earlier, an operation is an abstraction for the different tasks undertaken by a component. These operations are exposed through ports and can be requested by other components. When an operation is requested, the request is placed in the component's message queue and eventually serviced. When ready, the business logic of the operation i.e. a local callback is executed. This piece of code represents the brains of the operation. Our 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.

4.2.2 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.

4.2.3 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.

4.2.4 Evaluation of Solution

Evaluating the business logic grammar is important for two reasons. If the business logic modeling grammar is not sufficiently rich, then the model would not accurately represent the actual implementation code and this breaks any results obtained by CPN analysis. Secondly, preprocessing the business logic models helps system integrators analyze component interactions for unsafe interaction patterns e.g. cyclic dependencies, potential deadlocking behaviors etc. This implicitly constrains application developers about the types of

operational steps that are permitted when programming real-time component-based systems. Evaluating the accuracy of the business-logic model can be done by peer-review of models over a wide range of samples, compared against the implementation code to check for abstract correlation. The final step in the evaluating the correctness of such models involves in checking the expected execution times of each operational step. This is done by executing the component operation on the real-system while ensuring the absence of other interfering threads, and identifying the WCET of individual code blocks using monitoring methods e.g. logging.

4.2.5 Contributions

We have presented an approach [59] for modeling the operational behavior of each component in an application. The modeling grammar uses a sequence of timed steps that are executed by the operation, including interaction patterns. This approach enables abstracting the details of the middleware, while representing the temporal behavior of the component business logic.

4.3 Modeling and Analysis Improvements

4.3.1 Problem Statement

The CPN analysis work presented in [60] 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.

4.3.2 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.

4.3.3 Evaluation of Solution

Drastic changes to our CPN analysis e.g. representation of time, becomes risky when such changes are accompanied by unexpected execution behaviors. Enforcing time jumps requires strict semantic rules that need to be carefully judged e.g. jumping in time to the next timer expiry t_e on Hardware BBB_1 should not happen until all foreseen executions, interactions and triggers on all other nodes $BBB_2 ... BBB_n$ within the time frame t_{now} to t_e are executed. Following such rules on each hardware node, and on each calculated time jump leads to a safer execution. This is required because erroneous time jumps can lead to a crippled CPN model with inconsistent execution traces. Evaluating analysis improvements like time jumps requires large sets of unit tests with various types of examples, enforcing every time jump rule. Identifying erratic execution patterns leads to an improved set of rules and a more robust analysis model.

4.3.4 Contributions

Such issues are resolved with our analysis improvements, reported in [59]. 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 [56]. These improvements are evaluated in [59], where we detail our solutions with motivating examples.

4.4 Investigating Advanced State Space Analysis Methods

4.4.1 Problem Statement

State space analysis techniques have been successfully applied with Colored Petri Nets in a variety of practical scenarios and industrial use cases [46], [48]. 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.

4.4.2 Outline of Solution

The variety of CPN-specific state space reduction techniques [19], [45] 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 [108] analysis tool. The tool provides for several search algorithms and state space reduction techniques such as the *sweep-line method* [20] 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 [20] 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 [43] is that a complete state space search is guaranteed.

4.4.3 Evaluation of Solution

We try to 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 needs to be checked with negative test cases where the design model is known to be flawed.

4.4.4 Contributions

These methods were evaluated on our CPN analysis method and the results were presented in [59]. 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.

4.5 Experimental Validation of Timing Analysis Results

4.5.1 Problem Statement

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 [57] to simulate and analyze resilient cyber-physical systems consisting of 32 Beaglebone Black development boards [1]. We have chosen the light-weight ROS [86] middleware layer and implemented our ROSMOD Component model [58] on top of it. This component model provides the same execution semantics and interaction patterns as our DREMS component model [77]. 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.

4.5.2 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.

4.5.3 Evaluation of Solution

Experimentally validating our timing analysis results requires some careful evaluation. Such evaluation should include the testing parameters e.g. operating system, thread scheduler clock *tick*, clock time difference between nodes in distributed deployments etc. These factors play an important role in the execution traces observed for the tested applications. Since we are deriving WCET for operational steps from experiments, the experimental tests should be performed in *sterile* environments i.e. no other interfering threads or interactions besides the tested application component threads. Secondly, the monitoring of such applications should be performed with minimum possible overhead e.g. cyclic buffers for logging. Thirdly, each test must be run multiple times to obtain an average set of resultant behaviors to compare our CPN results against. This is necessary because testing methodologies, especially using prototype hardware can never be exhaustive.

4.5.4 Proposed Contributions

We will use our resilient CPS testbed to run experimental deployments of a variety of component-based distributed real-time scenarios. By executing component threads individually and at real-time priority, an execution profile i.e. pure execution times of the individual code blocks in component operations is obtained. The above measurements are obtained using a low-overhead logging framework that hooks into the component middleware and uses C++'s chrono high-resolution timers to record time instances in nanoseconds. We have a Python log analysis script that parses the component instance logs, identifies the time stamps at which operations are enqueued, dequeued and completed to generate execution plots that illustrate the component behavior. We will translate these results into business logic models and generate CPN timing analysis models, one per deployment. We will perform state space analysis and verification on the generated CPN analysis models and obtained execution plots of specific traces. We will compare the CPN analysis plots with the plots obtained by post-processing real execution logs. This comparison should indicate that the analysis model is consistently describing and analyzing the scenarios and providing conservative but close results.

4.6 Modeling and Analysis of Long-Running Component Operations

4.6.1 Problem Statement

Our DREMS 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 DREMS component model does not support the *cancellation* of long-running component operations to service other highly critical operations waiting in the queue. Our goal is to model and analyze component-based systems that support long-running operations, with checkpoints, that enable the novel integration of AI-type algorithms into our design and analysis framework.

4.6.2 Challenges

We need to identify the semantics of a long-running component operation i.e. the scenarios under which the component operations scheduler cancels a 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 equally-spaced checkpoints, then the operation would execute one step at a time and the scheduler would wait for the nearest checkpoint to reach, before canceling the operation (if needed). 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 cancellation 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 canceled. Secondly, the modeled long-running computation semantics must be incorporated into our component model so that any analysis results obtained can be suitably validated.

4.6.3 Outline of Solution

We are modeling long-running computations just like any other component operation, as outlined in Section 4.2. However, in each long-running operation, we will include a synchronous *checkpoint step*. 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 canceling 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 cancels. The higher priority operation is then processed.

4.6.4 Evaluation of Solution

Similar to previous state space analysis work, evaluating this solution requires a careful study of both the modeling and analysis aspects involved. The semantics of the modeled long-running operation, and the changes to the component operation scheduling needs to be checked with an array of examples, enforcing various functional behaviors. The analysis should incorporate a variety of scheduling schemes and interaction patterns under which the long-running operation is canceled or run to completion.

4.6.5 Proposed Contributions

We are working on modeling long-running component operations with our CPN analysis model in order to facilitate AI-type search algorithms in our framework. We are also planning on analyzing distributed component-based applications with long-running computations, to identify the downtime costs and overheads caused by canceling and restarting such operations.

4.7 Modeling and Analysis of Cyber-Physical Systems (CPS)

4.7.1 Problem Statement

An interesting extension to this work would be in investigating the utility of the CPN analysis to physical systems. Systems that are characterized by sensors, and actuators where software blocks interact with physical environments. Special components called *I/O components* could periodically receive data from sensors; each sensor periodically publishing data at with fixed update rate. Here, the schedulability problem would have to worry about timing requirements like actuation frequency, sensor sampling frequency etc. In such physical systems, the interaction patterns with which components and I/O devices communicate directly affect the timing properties of the system.

4.7.2 Challenges

The challenge with timing analysis of CPS lies in validating the analysis. As described earlier, there are several analysis methodologies in literature including prototypical testing and simulations. The quickest means to testing CPS applications with a prototype is to establish a realistic physics simulation engine in the loop. Component-based applications executing on realistic hardware and communicating with a high-fidelity system simulation environment creates a reasonable testing framework for CPS software. However, the challenge here lies in the simulation-in-the-loop. The software used by components to communicate with physics simulations varies from one simulation to another. The interaction patterns and threading behavior on the simulation server directly affects the execution of the component operations e.g. blocking times, jitter in sensor data reception etc. The true challenge in this problem is identifying the types of interactions that are commonly prevalent and modeling these interactions.

Once I/O components and devices are modeled, the analysis can verify for schedulability of the multi-rate multi-component CPS. Evaluating and validating these analysis results requires a prototype of the designed CPS. Our testbed [57] will again be used for

this purpose. We will use common physics simulators such as Kerbal Space Program [2], and Orbiter [3] to provide a periodic channel with updating sensor information, each sensor pertaining to a physics part in the simulation. The Beaglebone black development boards will execute the component-based software. I/O components will interact with the physics simulator and other regular components to control the CPS.

4.7.3 Outline of Solution

Our solution relies on our existing CPN analysis framework. Sensors in CPS are modeled as *periodic data pumps* i.e. data publishing entities that can be accessed by special components called *I/O Components*. Here, we do not model the data being published but merely it's rate. Actuators are modeled as time consuming non-blocking I/O device interactions i.e. when a component commands an actuator, the command takes a finite amount of WCET to complete. Therefore, the act of actuation is modeled as a simple operational step in our CPN model. Once we have this simple framework, we can model more complicated interactions with I/O devices e.g. polling interactions, synchronous blocking sensor data requests etc.

4.7.4 Evaluation of Solution

Validating our results for CPS systems will be harder than purely software-based systems. This is because we cannot build a testing prototype for every CPS being analyzed. We are therefore forced to rely on physics simulation engines, integrated in a control loop with component-based applications. This integration results in a cheap but effective CPS analysis workflow but also presents flaws e.g. it will be hard to ensure a stable and periodic sensor data pump in a physics simulation since the simulation is also a piece of software. Graphical load on the CPU, threading and concurrency issues can cause the physics simulation to not maintain its rate and therefore any observed results in a real-system may become useless. Thus, evaluating our CPN analysis results also requires a stable testing

environment with the chosen physics simulators. When such an environment is achieved, our basic testing and evaluation rules, as described in Section 4.5.3 apply.

4.7.5 Proposed Contributions

We will model component-based cyber-physical systems with our CPN analysis model. This requires us to model I/O devices and associated interaction patterns with software components. We will evaluate the modeling and perform state space analysis on the composed system. The obtained results will then be evaluated by comparison against prototype CPS deployments on our testbed. The generated real-world execution logs will be compared against our CPS analysis trace logs for validation.

CHAPTER V

CONCLUSIONS

In this proposal, we have described the class of distributed real-time embedded software systems we are addressing. We have provided detailed descriptions and reviews of relevant related work, covering a wide range of analysis tool suites. We have also briefed about the DREMS infrastructure, the backbone of the proposed research and development. We have elaborated on the Colored Petri net-based timing analysis methodology, describing both the modeling aspects and analysis results. The subsequent sections describe some interesting heuristics and modeling changes that greatly improved our analysis results, especially for distributed deployment scenarios. The remaining challenges within this scope include an experimental validation of the presented work and potential extensions to model and analyze Cyber-Physical systems.

Table 1 provides the tentative timetable for the proposed research.

Table 1: Proposed Research Timetable

Thorough Experimental Validation	09/2015 - 11/2015
Modeling and Analysis of Long-Running Component Operations	10/2015 - 12/2015
Modeling I/O components and analyzing CPS	11/2015 - 01/2016
Dissertation Writing	10/2015 - 03/2016

Appendices

APPENDIX A

PUBLICATIONS

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

1.1 Workshop Papers

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

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

1.2 Conference Papers

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

- P. Kumar, W. Emfinger, and G. Karsai. A Testbed to Simulate and Analyze Resilient Cyber-Physical Systems. In *Proceedings of the IEEE Rapid System Prototyping*, RSP 2015, Amsterdam, Netherlands, 2015. IEEE

- W. Emfinger, P. Kumar, A. Dubey, W. Otte, A. Gokhale, G. Karsai. DREMS: A Toolchain for the Rapid Application Development, Integration, and Deployment of

Managed Distributed Real-time Embedded Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS@Work 2013, Vancouver, Canada, 2013. IEEE

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

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

1.3 Journal Papers

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

- Levendovszky, T., A. Dubey, W. R. Otte, D. Balasubramanian, A. Coglio, S. Nyako, W. Emfinger, P. Kumar, A. Gokhale, and G. Karsai. DREMS: A Model-Driven Distributed Secure Information Architecture Platform for Managed Embedded Systems. In *IEEE Software*, vol. 99: IEEE Computer Society, 2014. IEEE

1.4 Book Chapters - Awaiting Reviews

- W. Emfinger, P. Kumar, A. Dubey, G. Karsai. Towards Assurances in Self-Adaptive, Dynamic, Distributed Real-time Embedded Systems. In *Software Engineering for Self-Adaptive Systems: Assurances*, 2015.

REFERENCES

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

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

[3] Orbiter Space Flight Simulator. http://orbit.medphys.ucl.ac.uk/.

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

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

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

[7] 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.

[8] 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.

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

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

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

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

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

[14] 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.

[15] B. W. Boehm. A spiral model of software development and enhancement. *Computer*,

21(5):61–72, 1988.

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

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

[18] 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.

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

[20] 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.

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

[22] 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.

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

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

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

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

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

[28] 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.

[29] 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.

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

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

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

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

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

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

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

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

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

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

[40] 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.

[41] 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.

[42] 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.

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

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

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

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

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

[48] 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.

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

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

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

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

[53] 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.

[54] 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.

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

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

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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

[63] 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.

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

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

[66] 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.

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

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

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

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

[71] 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.

[72] 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.

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

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

[75] 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.

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

[77] 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.

[78] 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.

[79] 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.

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

[81] 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.

[82] 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.

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

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

[85] 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.

[86] 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.

[87] 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.

[88] 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.

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

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

[91] 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.

[92] 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.

[93] 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.

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

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

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

[97] 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.

[98] 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.

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

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

[101] VisualStudio-2013-Documentation. *Run-time Text Generation with T4 Text Templates*, 2012.

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

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

[104] 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.

[105] 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.

[106] 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.

[107] 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.

[108] 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.

[109] 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.

[110] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal*

on Software Tools for Technology Transfer (STTT), 1(1):123–133, 1997.

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

[112] 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.

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

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

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