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

By

Pranav Srinivas Kumar

Proposal

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

Approved: Date:

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Real-time systems [51] are systems that 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. It is therefore paramount that real-time systems are analyzed early in the design process. Timing and schedulability analysis in software 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) [76] or response times [40]. 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.

Safety and mission-critical distributed real-time embedded (DRE) systems are increasingly being used in a variety of domains such as avionics [14], locomotive control [79], and industrial control systems [80]. 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 [12, 18, 30] 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.

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?

Also, component-based applications are typically executed on top of a large software stack, including a base operating system, a middleware communication layer, and a component model on which application processes execute. Analyzing such systems requires restraint on the modeling aspects as not all entities in this stack are necessarily relevant in deriving an *abstract model* of the system. When using formal analysis methods, the analysis model needs to be very abstract, capturing only crucial aspects of the design that is to be analyzed. Along with the formal model, a formal specification of timing requirements needs to be produced so that the analysis results can be appropriately interpreted. If the analysis model is accurate and the requirements are properly specified, then the derived results would verify the system design. Lastly, the verified result needs to be *validated*. Here, validation refers to informal analysis of the system, typically, by obtaining a prototype of the design and evaluating its performance. The observed behavior does not represent all possible behaviors that the system could exhibit but instead a candidate. Ideally, the execution trace observed in the real system is a subset of the execution traces analyzed by formal verification.

The most critical challenge using formal verification, especially in distributed systems is the large number of potential execution behaviors. The *state space* of the system i.e. the tree of possible execution traces that a system can reach from an initial state, can *explode* depending on the states in which individual entities in the model can exist. Complex systems typically exhibit a large state space of possible executions due to both the number of components and the individual behaviors. Verifying such a system requires that all of these execution traces be checked for inconsistency and timing violations. The challenge here is to establish an analysis model that enforces execution heuristics that tackle this state space explosion so that the overall number of states to be searched is greatly minimized. However, such heuristics are applicable only to a certain class of systems e.g. symmetry in deployment, and the explosion is difficult to avoid.

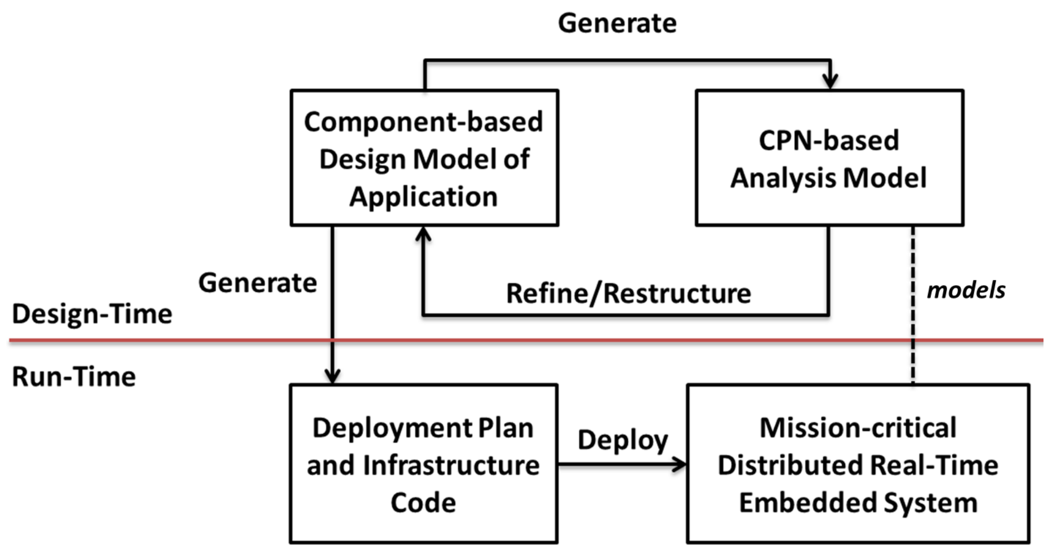

Generate
Component-based Design Model of Application
CPN-based Analysis Model
Generate
Refine/Restructure
models
Design-Time
Run-Time
Deployment Plan and Infrastructure Code
Deploy
Mission-critical Distributed Real-Time Embedded System

Figure 1: Verification-driven Workflow

The analysis work proposed in this thesis supports a verification-driven workflow for component-based software development, as shown in Figure 1. 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) [37] *formal analysis model* is generated. The system behavior is analyzed to check properties like lack of deadlocks, deadline violations etc. 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 has be re-verified at each stage to ensure the timing guarantees advertised.

The remainder of this proposal is organized as follows

- Chapter II describes the related research in timing analysis and verification for distributed real-time embedded applications.
- Chapter III introduces the DREMS infrastructure and Component Model used to experiment with and validate the timing analysis results.
- Chapter IV elaborates on the Colored Petri net-based timing analysis methodology devised for component-based DRE systems. This chapter includes 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

RELATED RESEARCH

2.1 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 simulation, prototype testing and formal verification.

2.1.1 Simulation

System simulation is a commonly used, often automated technique in the industry for testing control systems and algorithms [31, 41]. Simulations, although useful in exposing erratic behaviors, do no generally provide a definitive answer to verification queries i.e. the absence of a certain condition e.g. deadlocks, cannot be guaranteed. 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.

2.1.2 Testing with Prototypes

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.

2.1.3 Formal Analysis Methods

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 [7, 15]. 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: Deductive Reasoning and Model Checking.

Deductive Reasoning like theorem proving techniques formalize the system behavioral requirements into mathematical theorems. By proving mathematical theorems, system-level requirements are verified to hold. 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 [13], 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) [26]. 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 [29], and the Prototype Verification System (PVS) [58]. 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.

Model Checking methods, unlike theorem provers, are more suitable as debugging tools. Model checking is a technique to check the correctness of a system design by traversing the state space of the system while checking for errors. If an *error state*, a state with property violations, is *reachable* from an initial state, then the system design is veritably flawed. Such checks are typically expressed using temporal logic formula or similar state space queries that are evaluated at each state in the state space. Model checking methods are automatic verification techniques i.e. once the process begins, no human intervention is required. Model checking, as mentioned earlier, has been commonly used in the industry verifying communication protocols and hardware architectures. 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 [34], UPPAAL [48] and Kronos [77].

2.2 Petri net-based Timing Analysis for Concurrent Systems

Petri nets (PNs) [61] are a graphical modeling tool used for describing and analyzing a wide range of systems. A Petri net is a five tuple (P, T, A, W, M0), 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 M0 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 [20, 32], 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 [32] and timed systems [81]. For a detailed study of Petri nets and its applications, the reader is referred to standard textbooks [61, 65] and survey papers [54, 78, 82].

Petri nets have evolved through several generations from low-level Petri nets for control systems [65] to high-level Petri nets for modeling dynamic systems [39] to hierarchical and object-oriented Petri net structures [21] 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 [74], Stochastic Petri nets [10, 52] 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.

2.3 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 [67]. Development tools like MARTE [55] and AADL [25] 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.

2.3.1 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 [67], 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 [27] 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. Figure 2 shows the Symmetric Petri net derived from the AADL threading automata. 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.

The transformation rules for converting AADL models to Symmetric Petri nets presented in this work [67], are applied to a simple Producer-Consumer example (Figure 3). The example is composed of two processes, each containing two threads: a producer and a

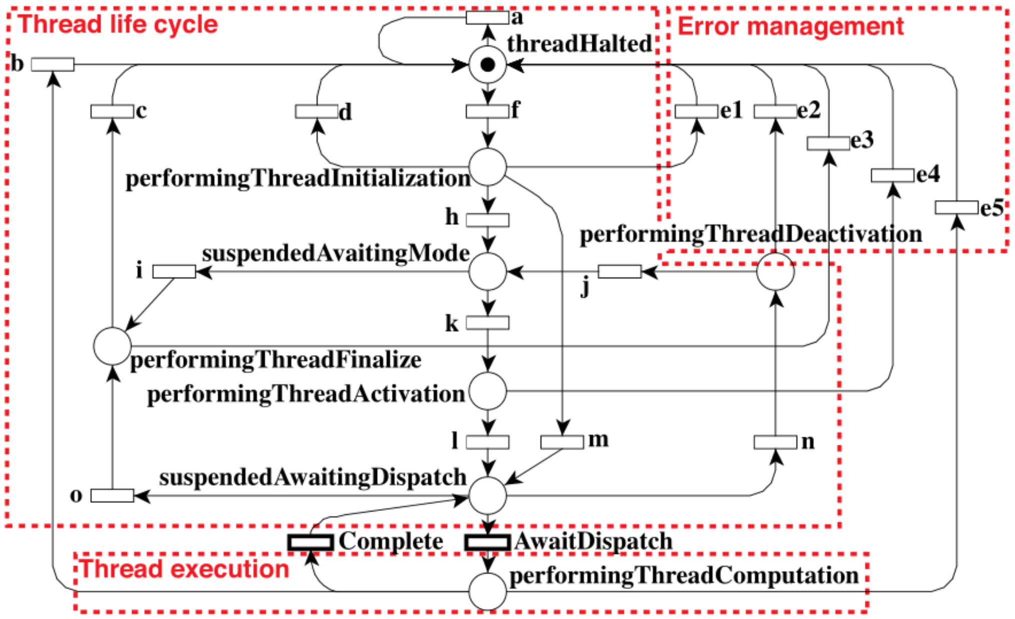


Figure 2: Petri net derived from an AADL Thread Automata, reprinted from [67]

consumer. The producer in the first process periodically communicates with the consumer in the second process and vice-versa.

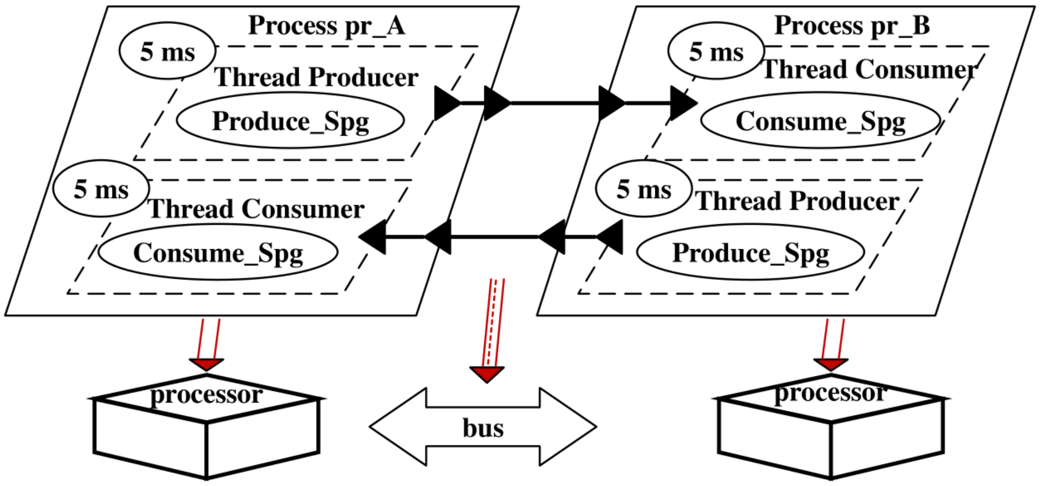


Figure 3: AADL Model of a Producer-Consumer Scenario, reprinted from [67]

Figure 4 shows the generated Petri net for this problem. The generation is automated using Ocarina [49], an AADL compiler. 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.

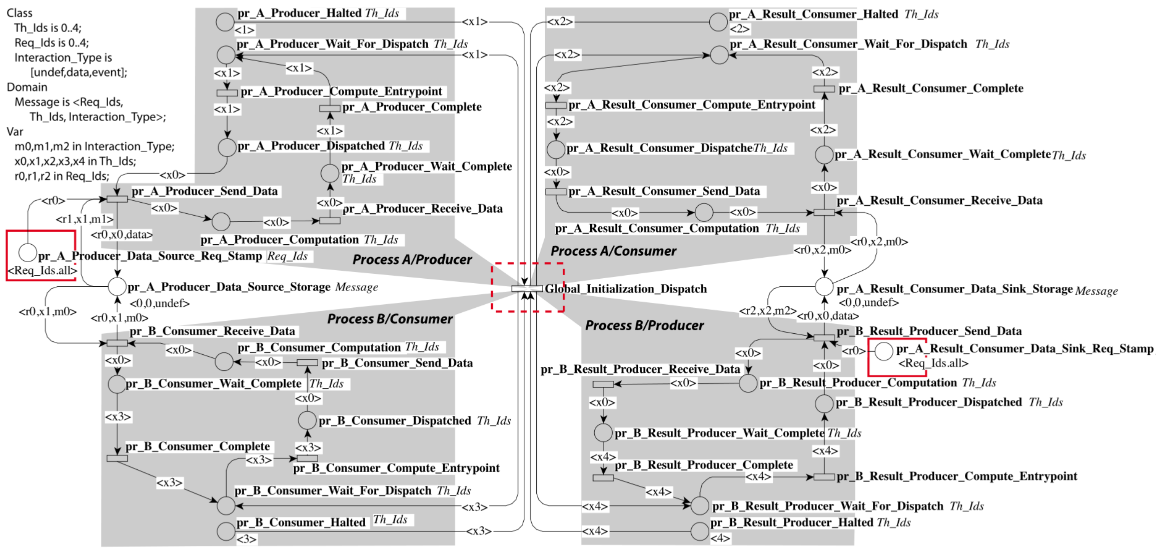

Class
Th_Ids is 0..4;
Req_Ids is 0..4;
Interaction_Type is [undef,data,event];
Domain
Message is <Req_Ids, Th_Ids, Interaction_Type>;
Var
m0,m1,m2 in Interaction_Type;
x0,x1,x2,x3,x4 in Th_Ids;
r0,r1,r2 in Req_Ids;
Process A/Producer
Process A/Consumer
Process B/Consumer
Process B/Producer
Global_Initialization_Dispatch
pr_A_Producer_Halted Th_Ids
pr_A_Producer_Wait_For_Dispatch Th_Ids
pr_A_Producer_Compute_Entrypoint
pr_A_Producer_Complete
pr_A_Producer_Dispatched Th_Ids
pr_A_Producer_Wait_Complete Th_Ids
pr_A_Producer_Send_Data
pr_A_Producer_Receive_Data
pr_A_Producer_Computation Th_Ids
pr_A_Producer_Data_Source_Req_Stamp Req_Ids
<Req_Ids.all>
pr_A_Producer_Data_Source_Storage Message
<0,0,undef>
pr_A_Result_Consumer_Halted Th_Ids
pr_A_Result_Consumer_Wait_For_Dispatch Th_Ids
pr_A_Result_Consumer_Complete
pr_A_Result_Consumer_Compute_Entrypoint
pr_A_Result_Consumer_DispatcheTh_Ids
pr_A_Result_Consumer_Wait_Complete Th_Ids
pr_A_Result_Consumer_Send_Data
pr_A_Result_Consumer_Receive_Data
pr_A_Result_Consumer_Computation Th_Ids
pr_A_Result_Consumer_Data_Sink_Storage Message
<0,0,undef>
pr_A_Result_Consumer_Data_Sink_Req_Stamp
<Req_Ids.all>
pr_B_Consumer_Receive_Data
pr_B_Consumer_Computation Th_Ids
pr_B_Consumer_Send_Data
pr_B_Consumer_Wait_Complete Th_Ids
pr_B_Consumer_Dispatched Th_Ids
pr_B_Consumer_Complete
pr_B_Consumer_Compute_Entrypoint
pr_B_Consumer_Wait_For_Dispatch Th_Ids
pr_B_Consumer_Halted Th_Ids
pr_B_Result_Producer_Send_Data
pr_B_Result_Producer_Receive_Data
pr_B_Result_Producer_Computation Th_Ids
pr_B_Result_Producer_Dispatched Th_Ids
pr_B_Result_Producer_Wait_Complete Th_Ids
pr_B_Result_Producer_Complete
pr_B_Result_Producer_Compute_Entrypoint
pr_B_Result_Producer_Wait_For_Dispatch Th_Ids
pr_B_Result_Producer_Halted Th_Ids

Figure 4: Producer-Consumer Petri net, reprinted from [67]

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 [70]. Lastly, the modeling constructs used are strictly bound to AADL and cannot be easily modified for systems not modeled using AADL.

2.4 Analyzing AADL Models with Timed Petri nets

The authors in [67], have also investigated AADL model analysis using Petri net extensions such as Timed Petri nets [66]. 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.

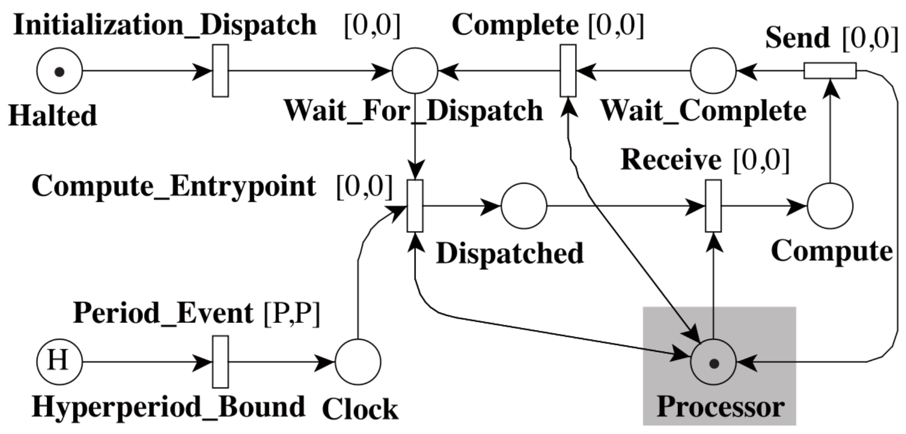


Figure 5: TPN Thread Pattern, reprinted from [66]

The approach presented in this work uses a *Timed Petri net pattern* to model the thread life-cycle, derived from the corresponding AADL model. Figure 5 shows the thread life cycle pattern. 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.

2.4.1 Deadline Violations

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 in Figure 5. Figure 6 shows the *monitoring* places that observe the thread execution to mark and record deadline violations any time the thread execution time exceeds its deadline.

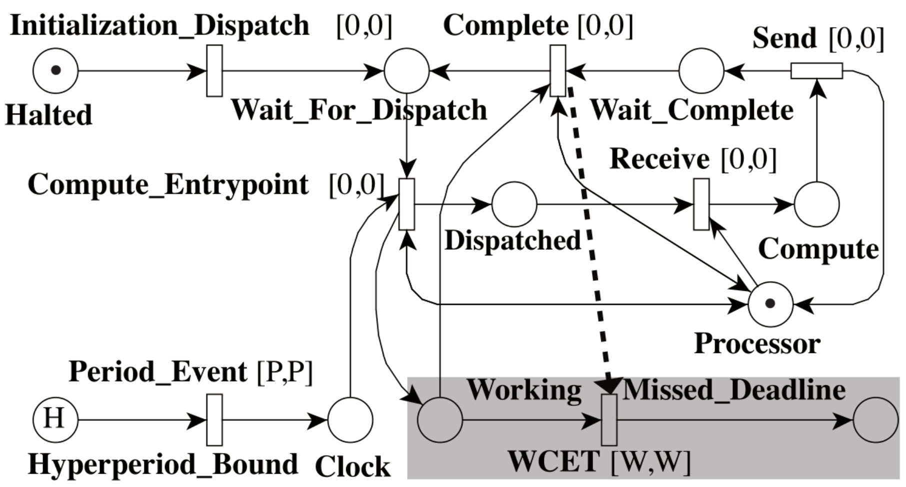


Figure 6: TPN Deadline Violation Detection, reprinted from [66]

2.4.2 Missed Activations

Similar to missed deadlines, missed activations can also be detected, as shown in Figure 7. Transition *Activation Time* is immediate, and is fired if the current thread is still working when a new Clock event is produced. When the thread must be dispatched but misses its activation deadline, the Activation Time transition fires, marking a missed activation.

When model checking this system, if there is no token in the *Missed Activation* place any where in the state space of the system, then no thread activations were missed.

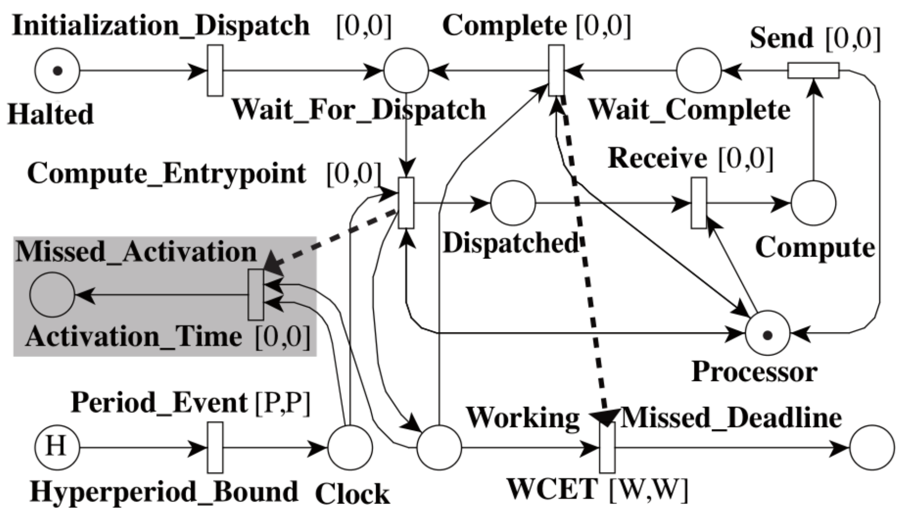


Figure 7: TPN Missed Activations, reprinted from [66]

Similar to this work, our Colored Petri net-based analysis work uses bounded observer places [4] that observe the system behavior for property violations and prompt completion of operations. However, this work [66] 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.

2.5 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 [57] 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.

2.5.1 Language Overview

MoSsRT is a domain-specific language offering functionality for design of hardware and software components that make up a real-time system. An operational layer describes the hardware model, an architectural representation of the software, and a behavior model describing software operation. It also covers system properties e.g. temporal properties. The language has several rules that enforce structural correctness of the models. The approach is sequential in nature. Once the model of the system is verified for correctness, temporal properties are analyzed for common use case problems.

2.5.2 Analysis Repository

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.

Figure 8, taken from [57], shows the array of functionality that MoSaRT provides. Note the bidirectional transformation between the design model and the intermediate 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.

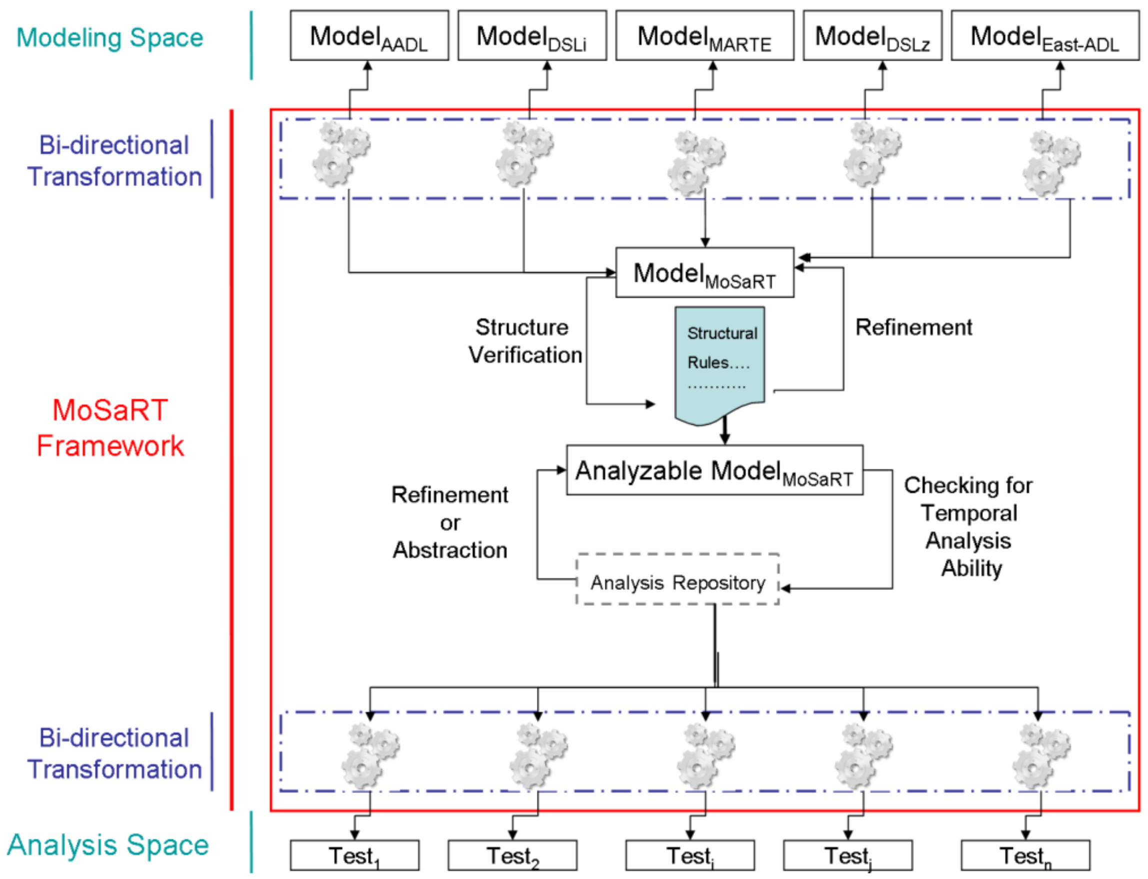


Figure 8: MoSaRT Framework, reprinted from [57]

2.6 MAST: Modeling and Analysis Suite

MAST [28] 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 [42, 50] and distributed real-time systems [60, 71].

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 [53].

For analysis, the MAST suite includes schedulability analysis tools that use newly published research techniques such as the offset-based methods [59] that enhance analysis results, providing less pessimistic estimates than previous results [71]. Figure 9 shows the MAST tool suite. The system description is specified through an ASCII description language that serves as the input to the analysis tools.

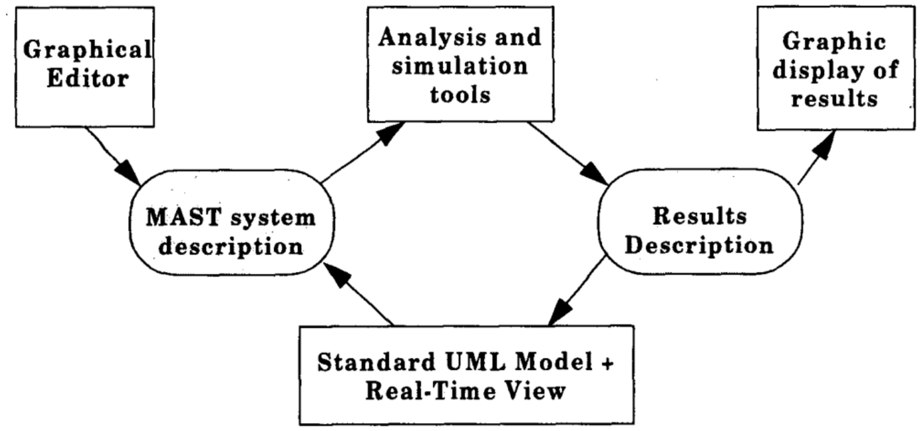

Graphical Editor
Analysis and simulation tools
Graphic display of results
MAST system description
Results Description
Standard UML Model + Real-Time View

Figure 9: MAST Environment, reprinted from [28]

Using UML, the real-time *view* of the DRE is described [68] 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 9 like the graphical editor and results display interfaces are not available or incomplete.

2.6.1 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. Figure 10 shows an example transactional view of a system execution.

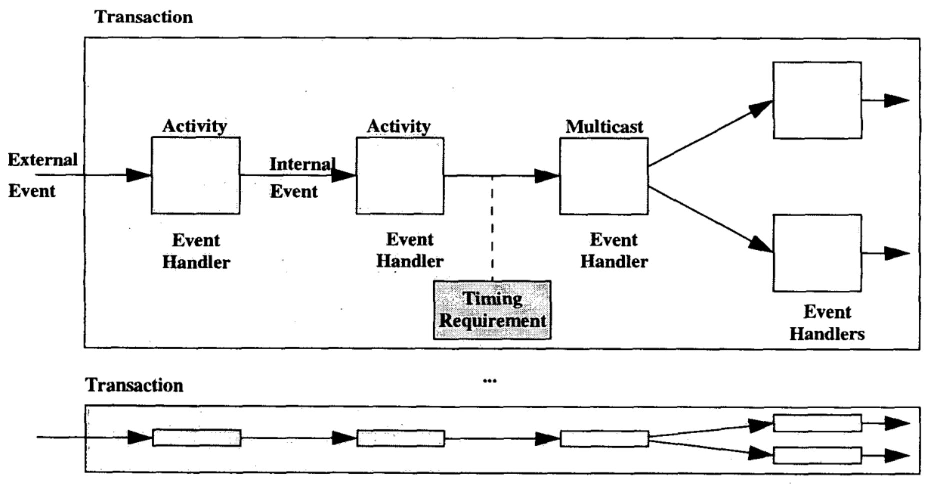


Figure 2. Real-Time System composed of transactions

Figure 10: MAST Transactional Model, reprinted from [28]

This transactional view shows 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).

2.6.2 Model Generation and Analysis

The MAST analysis methodology is described [28] with the following example. The example, as shown in Figure 11, is a simplified control system of a teleoperated robot - a distributed system with two specialized nodes: a robot controller and a remote teleoperation station.

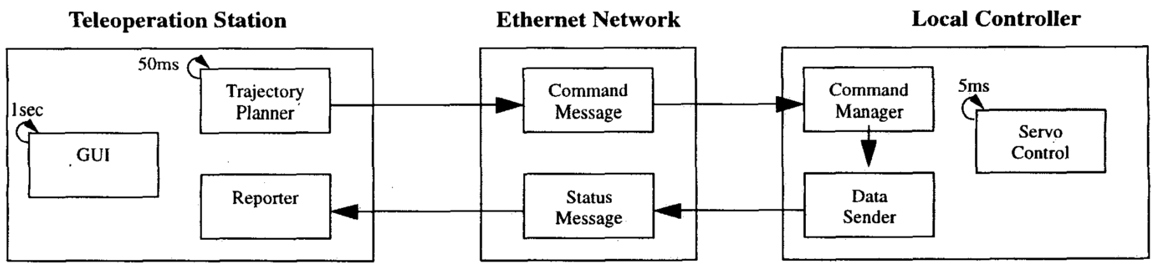
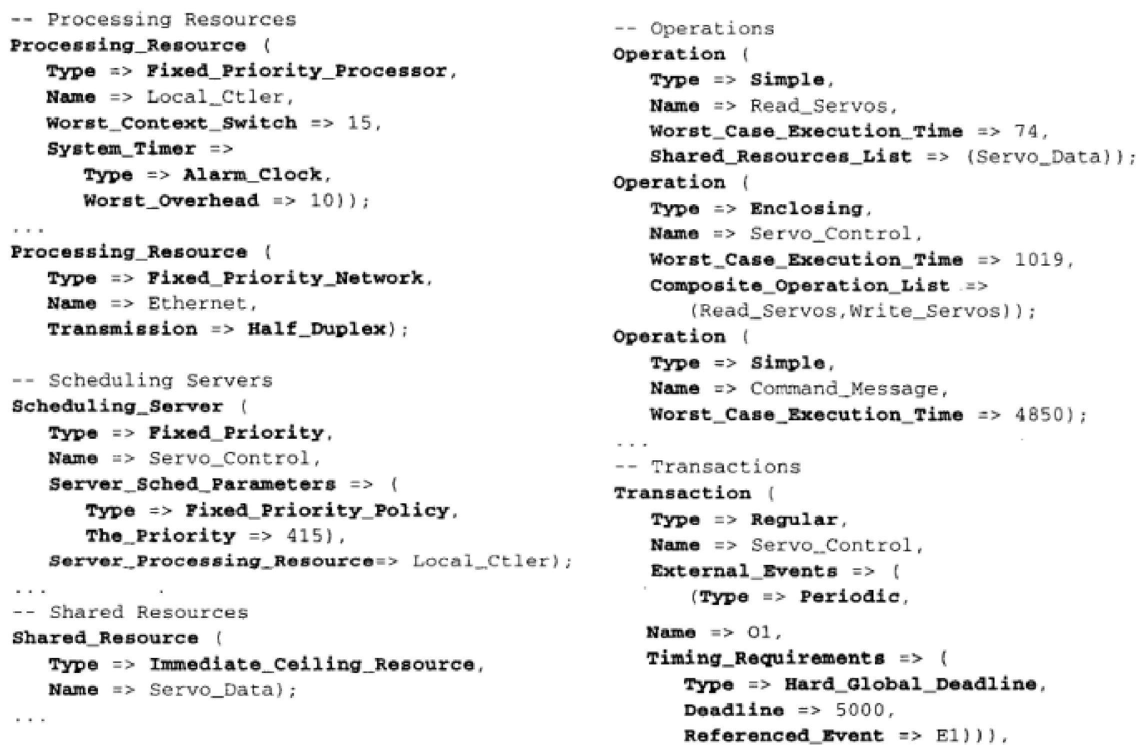


Figure 11: Teleoperated Robot, reprinted from [28]

Figure 12 shows a segment of the generated MAST description for this system. In order, the processing resources, the scheduling servers, shared resources, operations, and transactions are described. Timing requirements are described in all transactions. Scheduling parameters are embedded in each scheduling server. This model is fed to the MAST worst-case analysis tools to obtain results on worst-case response times of all activities, which are then compared with the transaction deadlines to determine schedulability.



```
-- Processing Resources
Processing_Resource (
   Type => Fixed_Priority_Processor,
   Name => Local_Ctler,
   Worst_Context_Switch => 15,
   System_Timer =>
      Type => Alarm_Clock,
      Worst_Overhead => 10));
...
Processing_Resource (
   Type => Fixed_Priority_Network,
   Name => Ethernet,
   Transmission => Half_Duplex);

-- Scheduling Servers
Scheduling_Server (
   Type => Fixed_Priority,
   Name => Servo_Control,
   Server_Sched_Parameters => (
      Type => Fixed_Priority_Policy,
      The_Priority => 415),
   Server_Processing_Resource=> Local_Ctler);
...
-- Shared Resources
Shared_Resource (
   Type => Immediate_Ceiling_Resource,
   Name => Servo_Data);
...
```

```
-- Operations
Operation (
   Type => Simple,
   Name => Read_Servos,
   Worst_Case_Execution_Time => 74,
   Shared_Resources_List => (Servo_Data));
Operation (
   Type => Enclosing,
   Name => Servo_Control,
   Worst_Case_Execution_Time => 1019,
   Composite_Operation_List =>
      (Read_Servos,Write_Servos));
Operation (
   Type => Simple,
   Name => Command_Message,
   Worst_Case_Execution_Time => 4850);
...
-- Transactions
Transaction (
   Type => Regular,
   Name => Servo_Control,
   External_Events => (
      (Type => Periodic,

   Name => O1,
   Timing_Requirements => (
      Type => Hard_Global_Deadline,
      Deadline => 5000,
      Referenced_Event => E1))),
```

Figure 12: MAST Dsscription of Teleoperated Robot, reprinted from [28]

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.

2.7 Some Common Timing Analysis Tools

2.7.1 Cheddar Real-Time Scheduling Framework

Cheddar [69] 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 [50]: 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.

2.7.2 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 [11, 19, 48] 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 [5], 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. An overview of the UPPAAL is shown in Figure 13. 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.

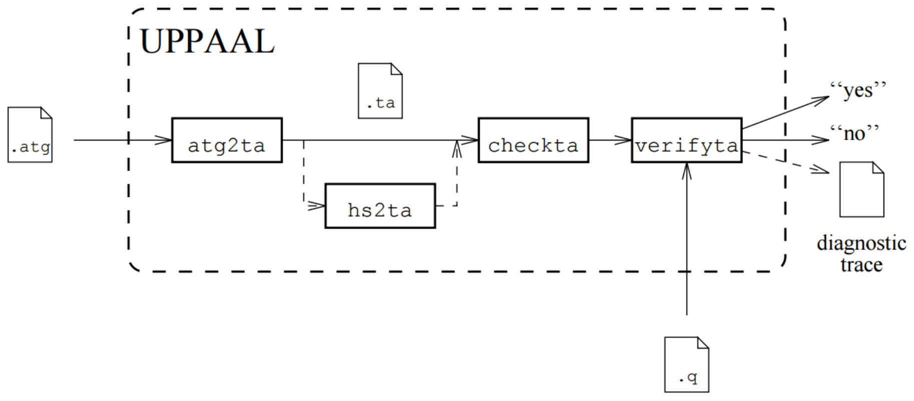


Figure 13: UPPAAL Architecture, reprinted from [48]

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.

2.7.3 TIMES

TIMES [6] 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.

CHAPTER III

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

3.1 Context: 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) [23], [56]. 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 [22] that integrate with a well-defined and fully implemented component model [47, 56] 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.

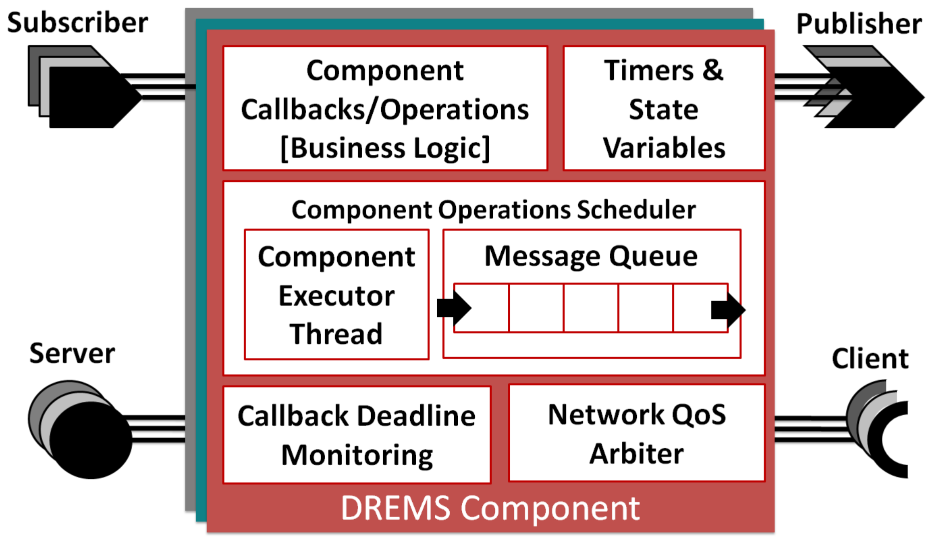

Subscriber
Publisher
Component Callbacks/Operations [Business Logic]
Timers & State Variables
Component Operations Scheduler
Component Executor Thread
Message Queue
Server
Client
Callback Deadline Monitoring
Network QoS Arbiter
DREMS Component

Figure 14: DREMS Component

Figure 14 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.

DREMS Components interact via anonymous publish/subscribe [24] message passing and peer-to-peer synchronous and asynchronous remote method invocation (RMI and AMI) [63, 73] mechanisms. Component interfaces expose operations that can be invoked/requested by several agents including time-triggered events, requests from outside components and requests from the underlying communication infrastructure. 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 interactions like this, every operation request coming from an external entity reaches the component through its message queue. This queue is maintained by the component-level scheduler that schedules operations for execution. When ready, a single component executor thread per component will be released to execute the operation requested by the front of the component message queue. Scheduling schemes in the message queue include FIFO (first-in first-out), PFIFO (priority FIFO) and EDF (earliest deadline first). The chosen operation runs to completion before the next request can be processed. Therefore, the component operation scheduling is non-preemptive and single threaded. Note however that multiple components can be executed concurrently, leading to schedulability challenges and motivating timing analysis.

On the operating system level, DREMS components are grouped into processes that may be assigned to ARINC-653 [8] styled temporal partitions, implemented by the DREMS OS scheduler. 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 and assigns a guaranteed slice of the CPU's time to that partition. The repeating partition windows are called minor frames. The aggregate of minor frames is called a major frame. The duration of each 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 duration of a partition defines the amount of time available per hyperperiod to schedule all threads assigned to that partition. 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 presented 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.

3.2 Problem Statement

Consider a set of mixed-criticality component-based applications, distributed and deployed across a cluster of embedded computers. Each component exposes a set of interfaces to other components in the cluster and also to the underlying infrastructure. Once deployed, each component executes operational requests received on its message queue. Each component is associated with a single executor thread that handles these requests. This executor thread is scheduled concurrently with a known set of highly critical system threads and other low priority threads. Furthermore, the application threads may be subject to a temporally partitioned scheduling scheme.

Assumptions about the system include the following:

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 the above assumptions, the high-level problem here is to ensure that the temporal behavior of all the application components lies within the bounds laid out by the system specifications.

3.3 Modeling Component Operation Business Logic: COMPLETED

3.3.1 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.

3.3.2 Contributions

We have presented an approach [46] for modeling the operational behavior of each component in an application. The model uses a sequence of timed steps that are executed by a component operation, including steps the include interactions with other components. This approach enables abstracting the details of the middleware, while representing the temporal behavior of the component business logic.

Figure 15 shows the Extended Backus-Naur form representation of the grammar used for modeling the business logic of component operations. The symbol ID represents identifiers, a unique grouping of alphanumeric characters/terminal symbols and the symbol INT

```
(* Business Logic syntax in Extended Backus-Naur Form *)
business_logic          =     ‘Do’, ws, operation_name, ws
                              ‘[’, operation_priority, operation_deadline, ‘]’, ‘{‘, { functional_step }, ‘};’ ;
operation_name          =     ID ;
operation_priority      =     INT ;
operation_deadline      =     INT ;
functional_step         =     {sequential_code_block | rmi_call | ami_call | dds_publish | dds_pull_subscribe |
                                            dds_push_subscribe | loop} ;
sequential_code_block =       INT, ‘;’ ;
rmi call                =     ‘RMI’, ws, receptacle_port, ‘.’, remote_operation, ‘[’ query_time, processing_time ‘];’ ;
ami call                =     ‘AMI’, ws, receptacle_port, ‘.’, remote_operation, ‘[’ query_time, processing_time ‘];’ ;
dds publish             =     ‘DDS_Publish’, ws, dds_port, ‘.’, topic, ‘[’, publish_time, ‘];’ ;
dds pull subscribe      =     ‘DDS_Pull_Subscribe’, ws, dds_port, '.‘, topic, '[‘, processing_time, ‘];’ ;
dds_push_subscribe      =     ‘DDS_Push_Subscribe’, ws, dds_port, ‘.’, topic, ‘[’ processing_time, ‘];’ ;
loop                    =     ‘LOOP’, ws, ‘[’, count, ‘]’, ws, ‘{‘, {functional_step}, ‘};’ ;
receptacle_port         =     ID ;
remote_operation        =     ID ;
dds_port                =     ID ;
topic                   =     ID;
query_time              =     INT ;
processing_time         =     INT ;
publish_time            =     INT ;
count                   =     INT;
ws                      =     ? white space characters ?
identifier          =         alphabetic character, { alphabetic character | digit } ;
Integer             =         digit, {digit} ;
digit               =         “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9” ;
```

Figure 15: DREMS Component Operations - Business-logic Metamodel

represents positive integer digits. Each operation is characterized by a unique name, a priority and a deadline. The priority is an integer used to resolve scheduling conflicts between operations provided by the same component when requests from external entities are received. The arbitration is handled by the component-level scheduler. The deadline of the operation is the worst-case amount of time that can elapse after the operation is marked as ready.

The business logic of every component operation is modeled as a sequence of functional steps, each with an assigned worst-case execution time. We broadly classify these steps into (1) blocks of sequential code, (2) peer-to-peer synchronous and asynchronous remote calls, (3) anonymous publish/subscribe distribution service calls, (4) blocking and non-blocking I/O interactions and (5) bounded control loops. Notice the integration of timing properties such as worst-case function call times (query_time), worst-case argument processing times (processing_time) and DDS publish times (publish_time). If these expected delays are set to zero, the analysis will execute these interactions in a single synchronous step taking no time. However, in reality these steps still take a non-zero amount of time to execute. Therefore, if such metrics are not known then these values can be set to zero and an overall worst-case execution time can be set per operation. This is the maximum amount of time that can elapse after the component operation has begun to execute. This time will include all component interactions and network delays that affect the operation's execution.

The following section describes a Colored Petri-net based modeling technique for classes of component-based systems that concisely models the various aspects of a distributed deployment. The purpose of the model is to not only be able to simulate system designs but also provide efficient means for state space analysis of large, complex and interacting systems. The business logic modeled in this section is incorporated in the Colored Petri net data structures and determines the sequence of execution steps observed when components interact.

3.4 Colored Petri net-based Modeling of Component-based Distributed Real-time Applications: COMPLETED

3.4.1 Challenges

The CPN model should capture the behavioral semantics of our component model described in [56], 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.

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.

3.4.2 The Choice of Colored Petri Nets

With Colored Petri Nets (CPN) [37], 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. Sections 3.6 and 3.7 cover in more detail how some of the modeling concepts/changes affect and improve the efficiency of the analysis.

3.4.3 Contributions

This section briefly describes how Colored Petri nets are used to build an extensible, scalable analysis model for component-based applications. Several nets are used to compose the different layers of the design model. To edit, simulate and analyze this model, we use the CPN Tools [64] tool suite.

The top-level CPN Model is shown in Figure 16. The places (shown as ovals) in this model maintain *colored* (typed) tokens that represent the states of interest for analysis. For instance, the *Clocks* place holds tokens of type *clock_tokens* maintaining information regarding the state of the clock values and temporal partition schedule on all computing nodes. To ensure modularity, this model is partitioned into two interacting sub-nets to handle the hierarchical scheduling.

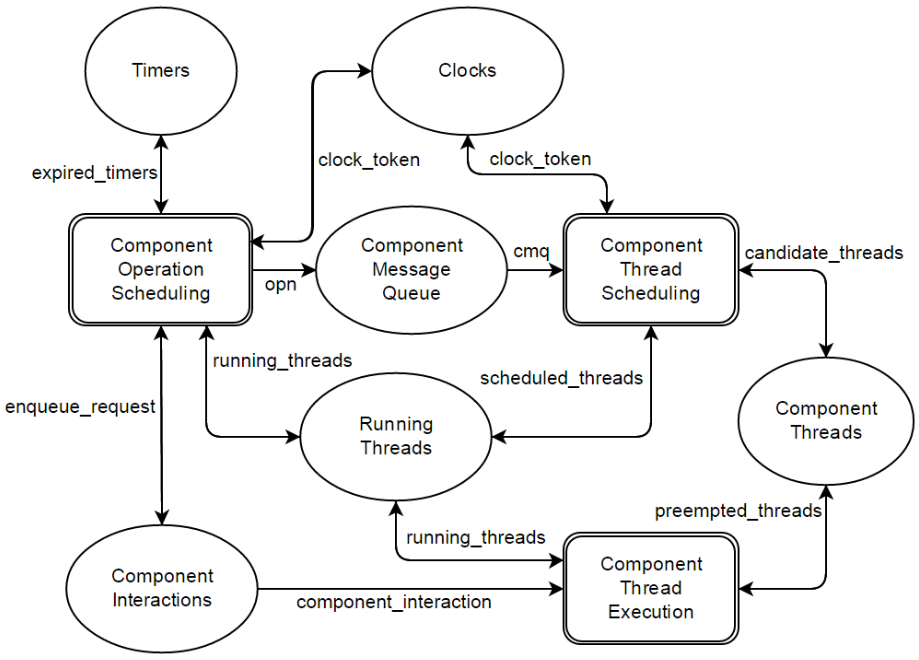


Figure 16: Top-level CPN Model

3.4.3.1 Component-level Execution Model

Component Operations: Every operation request O made on a component C_x is a *record* type implementation of the 4-tuple:

$$O(C_x) = < ID_O,\ Prio_O,\ Dl_O,\ Steps_O > \tag{1}$$

where, ID_O is a unique concatenation of strings that help identify and locate this operation in the system (consisting of the name of the operation, the component, the computing node, and the temporal partition). The operation's priority ($Prio_O$) is used by the analysis engine to enqueue this operation request on the message queue of C_x using a fixed-priority non-preemptive FIFO scheduling scheme. The completion of this enqueue implies that this operation has essentially been *scheduled* for execution. Once enqueued, if this operation does not execute and complete before its fixed deadline (Dl_O), its real-time requirements are violated.

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

$$s_i = < Port,\ Unblk_{s_i},\ Dur_t,\ Exec_t > \tag{2}$$

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

Component Interactions: Consider an application with two components: a client and a server. The client is periodically triggered by a timer to make an remote method call to the server. We know that when the client executes an instance of the timer-triggered operation, a related operation request is enqueued on the server's message queue. In reality, this is handled by the underlying middleware. Since we do not model the details of this framework, the server-side request is modeled as an *induced operation* that manifests as a consequence of the client-side activity. Tokens that represent such design-specific interactions are maintained in the place *Component Interactions* (Figure 16) and modeled as shown in equation 3. The interaction *Int* observed when a component C_x queries another component C_y is modeled as the 3-tuple:

$$Int(C_x, C_y) = < Node_{C_x},\ Port_{C_x},\ O(C_y) > \tag{3}$$

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

Timers: DREMS components are inactive initially; once deployed, a component executor thread is not eligible to run until there is a related operation request in the component's message queue. To start a sequence of component interactions, periodic or sporadic timers can be used to trigger a component operation. In CPN, each timer TMR is held in the place *Timers* and represented as shown in Eq. 4. Timers are characterized by a period (Prd_{TMR}) and an offset (Off_{TMR}). Every timers triggers a component using the operation request O_{TMR}.

$$TMR = < Prd_{TMR},\ Off_{TMR},\ O_{TMR} > \tag{4}$$

3.4.3.2 OS-level Thread Execution Model

Temporal Partitioning: The place *Clocks* in Figure 16 holds the state of the node-specific global clocks. The temporal partition schedule modeled by these clocks enforces a constraint: component operations can be scheduled and component threads can be run only when their parent partition is active. Each clock token *NC* is modeled as a 3-tuple:

$$NC = < Node_{NC},\ Value_{NC},\ TPS_{Node_{NC}} > \tag{5}$$

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

$$TP = < Name_{TP},\ Prd_{TP},\ Dur_{TP},\ Off_{TP},\ Exec_{TP} > \tag{6}$$

Each partition TP (Eq. 6) is modeled as a record color-set consisting of a name $Name_{TP}$, a period Prd_{TP}, a duration Dur_{TP}, an offset Off_{TP} and the state variable $Exec_{TP}$.

Component Thread Behavior: Figure 17 presents a simplified version of the CPN to model the thread execution cycle. The place *Component Threads* holds tokens that keep track of all the ready threads in each computing node. Each component thread *CT* is a record characterized by:

$$CT = < ID_{CT},\ Prio_{CT},\ O_{CT} > \tag{7}$$

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

If the highest priority thread is not already servicing an operation request, the highest priority operation from *Component Message Queues* is dequeued and scheduled for execution (held by O_{CT}). The scheduled thread is placed in *Running Threads*.

When a thread token is marked as running, the model checks to see if the thread execution has any effect on itself or on other threads. These state changes are updated using the transition *Execute Thread* which also handles time progression. Keeping track of $Value_{NC}$, the thread is preempted at each clock tick. This loop repeats forever, as long as there are no system-wide deadlocks.

This modeling paradigm was reported [47] 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. Analysis methods applicable to this

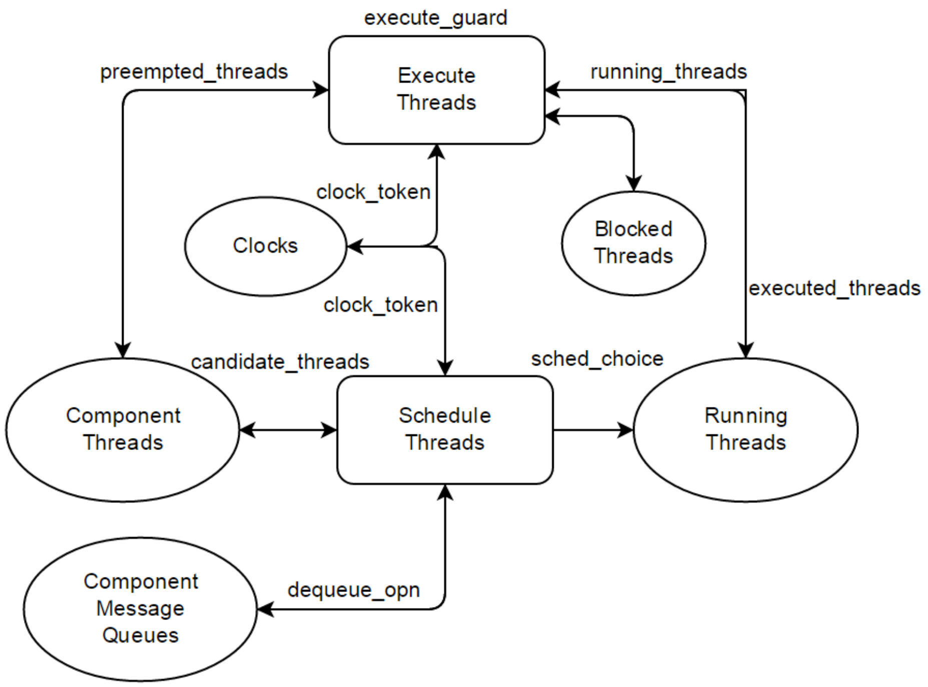


Figure 17: Component Thread Execution Cycle

example, along with early results are presented in Section 3.6. Improvements to the modeling concepts and scalability results obtained from this CPN methodology are discussed in Section 3.7.

3.5 Generating CPN Analysis Model from System Design Model: COMPLETED

3.5.1 Requirements

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 3.3 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.

3.5.2 Contributions

A *CPN Generator* model interpretor was developed to generated Colored Petri net analysis models from system designs. The interpreter parses the design model tree, identifies system properties such as component definitions, port properties, scheduling schemes etc. and constructs an interpreted tree. Using this tree, system properties are converted to relevant CPN token strings. Using T4 Text Templates [72] 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.

The interesting aspect of this CPN analysis is the fact that the structure of the model is largely unchanged when analyzing different deployments. By this, we mean that the places, transitions, arcs, arc guards, transition guards and functions are constant. Unlike the AADL to Petri net translation work [67], new sub-models need not be generated and composed for minute changes in the design model. All of the parameterized aspects of the Colored Petri net are encoded in its token structures. So, any changes to the design model causes new initial tokens that need to be generated into specific places in the net. This not only makes the model efficient and scalable but also makes the model generation simple.

3.6 State Space Analysis and Verification of Safety-critical System Properties: COMPLETED

Given a CPN model (that was generated from a component architecture and deployment model), a state-space of the system can be constructed using the semantics of CPN. This state space is infinite, however, in practice, it is often sufficient to consider some finite subset, starting from a initial state up to a few hyperperiods of the partition scheduler. In order to describe the utility of state space analysis, we consider a simple trajectory planning application (TPA). The following results were reported in [47].

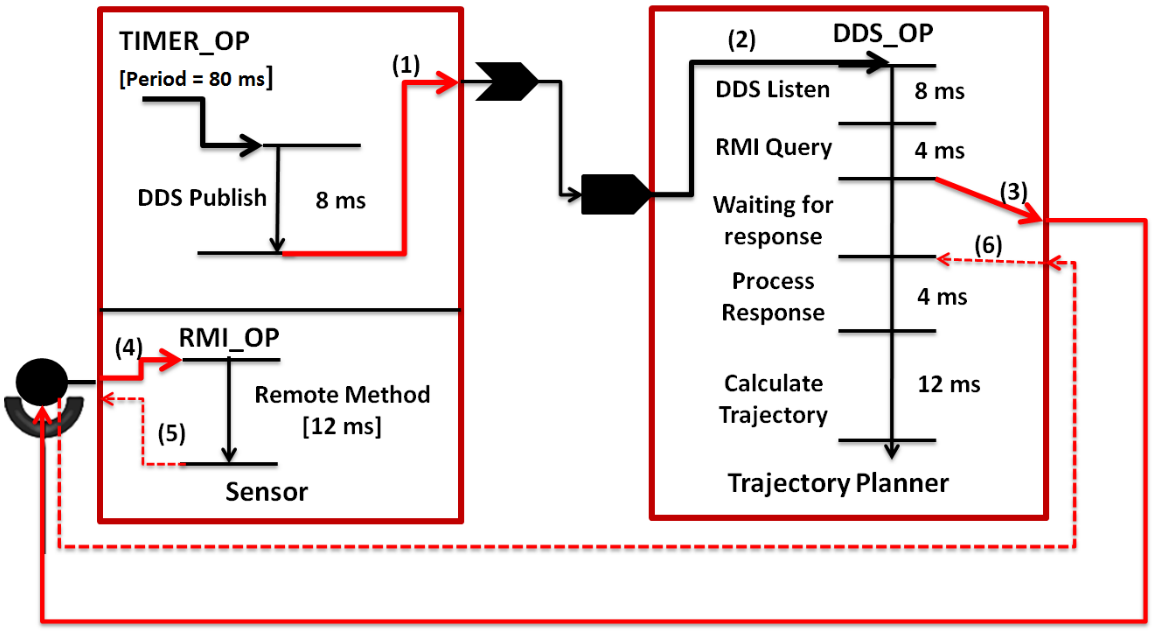


Figure 18: Trajectory Planner - Component Assembly

The component assembly for this application is shown in Figure 18. A Sensor component periodically publishes on a trigger topic, notifying the Trajectory Planner of the existence of new sensor data. Once the notification is received, the Trajectory Planner makes an RMI call to retrieve the data structure of sensor values, using which the satellite trajectory is updated. The sequence of steps in each of these operations is referred to as

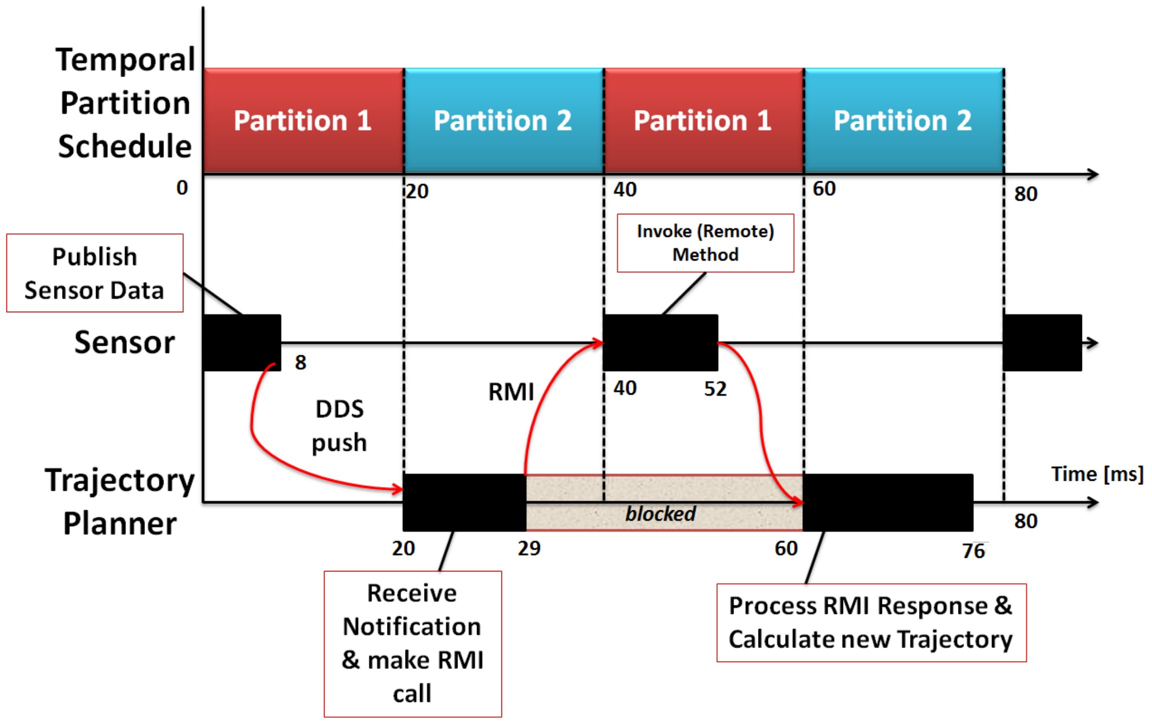


Figure 19: Trajectory Planner - Timing Diagram

the business logic of the operation. This business logic is modeled using our textual language in the modeling tools, in which the designer specifies the macro execution steps in a component operation along with worst-case estimated time taken on each step. Figure 19 shows an ideal-case timing diagram, where there are no other threads executing in the system.

The analyzable states of this system are observed in the markings of the various CPN places in the model. Using the built-in state space analysis in CPN Tools a bounded state space of the system is generated. Using both standard and user-defined queries, this state space is searched to check system properties like lack deadline violations and deadlocks, bounds on response times etc.

3.6.0.1 Deadline Violation Detection:

Each time a component operation is scheduled, the clock value of the node is recorded as the "start time" of the operation. If this operation is incomplete when the clock reaches the operation's deadline, a deadline violation is detected. Using the *SearchNodes* function in CPN Tools, the deadline violations on any component operation can be identified by observing all component operations each time the node-specific clock progresses. In Figure 19, the *DDS_OP* on the Trajectory Planner takes 56 ms to complete, measured from when the operation was enqueued and marked as ready. If the deadline of this operation is set to 50 ms, a state space search would reveal a deadline violation when the clock reaches 51 ms.

3.6.0.2 Worst-case Trigger-to-Response Time Calculation:

For a known trigger operation and desired response operation, the worst-case trigger-to-response time can also be calculated from the generated state space. Using the names of the trigger and response operations, a state space node that presents the earliest completion of the trigger operation and the latest completion of the response operation within the set period is identified. In the Trajectory Planning application, considering the *TIMER_OP* to be the trigger and the trajectory planning *DDS_OP* to be the response, the worst-case response time is found to be 68 ms (Trigger completes earliest at 8 ms and response completes latest at 76 ms).

3.6.0.3 Partial Thread Execution Order Generation:

In development scenarios where an application developer is aware of the operation-specific timing requirements but not thread priorities, the analysis is capable of identifying partial thread execution orders that satisfy the requirements. If all unknown thread priorities are set to a common value, the generated partial state space will then encapsulate the set of non-deterministic thread execution orders that arise from the scheduling. Using timing requirements of the form - *Once Operation A on Component A on Node A completes, Operation B on Component B on Node B must complete within 150 ms*, a state space node satisfying this requirement can be identified by querying the generated state space. A backtrace from this node enables assigning thread priorities to ensure the satisfaction of the timing requirement.

3.6.0.4 Scalability Testing:

The size of the generated state space is dependent on the amount of concurrency in the behavior. 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. This analysis model has been identified to scale well for medium-sized applications, tested up to 100 components distributed on up to 5 computing nodes. Table 1 summarizes these results.

Table 1: Scalability Results

Scenario	Nodes	Partitions / Node	Threads / Partition	Hyper-periods	State Space	Generation Time
TPA	5	2	1	10	180	0.981s
Sample2	2	5	5	10	124,469	14.1m
Sample3	5	5	4	10	485,552	36.5m

The following section describes some interesting heuristics and modeling methods applied to this CPN approach that dramatically improve the efficiency and utility of state space analysis for large distributed systems.

3.7 Modeling and Analysis Improvements: COMPLETED

The CPN analysis work presented in [47] 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 when the computers can be synchronously progressed. Such issues are resolved with our analysis improvements, reported in [46]. This work is presented below.

3.7.1 Handling Time

The CPN-based analysis consists of executing a simulation of the model and constructing a state space data structure for the system (for a finite horizon), and then performing queries on this data structure. This is automated by CPN Tools. The first improvement over the basic CPN approach is in how we handle time. Although it is true that CPN and similar extensions to Petri Nets such as Timed Petri Nets inherently have modeling concepts for simulation time, we explicitly model time as an integer-valued *clock* color token in CPN. There are several reasons for this choice.

Firstly, this is an extension to our previous arguments about choosing Colored Petri Nets. Modeling the OS scheduler clock as a colored token allows for extensions to its data structure such as (1) intermediate time stamps and internal state variables, and (2) adding temporal partitioning schemes like the (time-partitioned) ARINC-653 [8] scheduling model (Figure 20).

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

Figure 20: A Clock Token with Temporal Partitioning

These extended data structure fields can be more easily manipulated and used by the model transitions during state changes, allowing for richer modeling concepts that would not be easily attainable using token representations provided by Timed Petri Nets. The ability to pack colored tokens with rich data structures also reduces the total number of colors required by the complete model. This quantitative measure directly influences the reduced size of the resultant state space. The downside of this approach to modeling is that we have to choose a time quantum. But in practical systems this is usually not a problem, as the low-level scheduling decisions are taken by an OS scheduler based on a time scale with a finite resolution. We have chosen 1 msec as the quantum (corresponding to the typical 1KHz scheduler in Linux), but it can be easily changed.

Secondly, modeling time as a token allows for smarter time progression schemes that can be applied to control the pace of simulation. If we did not have such control over time, the number of states recorded for this color token would eventually explode and itself contribute to a large state space. In order to manage this complexity, we have devised some appropriate *time jumps* in specific simulation scenarios.

If the rate at which time progresses does not change, then for a 1 msec time resolution, S seconds of activity will generate a state space of size: $SS_{size} = \sum_{i=1}^{S*1000} TF_{t_i}$ where TF_{t_i} is the number of state-changing CPN transition firings between t_i and t_{i+1}. This large state space includes intervals of time where there is no thread activity to analyze either due to lack of operation requests, lack of ready threads for scheduling, or due to temporal partitioning.

During such idle periods, it is prudent to allow the analysis engine to *fast-forward* time either to (1) the next node-specific clock tick, (2) the next global timer expiry event, or (3) the next activation of the node-specific temporal partition (whichever is earliest and most relevant). This ensures that the generated state space tree is devoid of nodes where there is no thread activity.

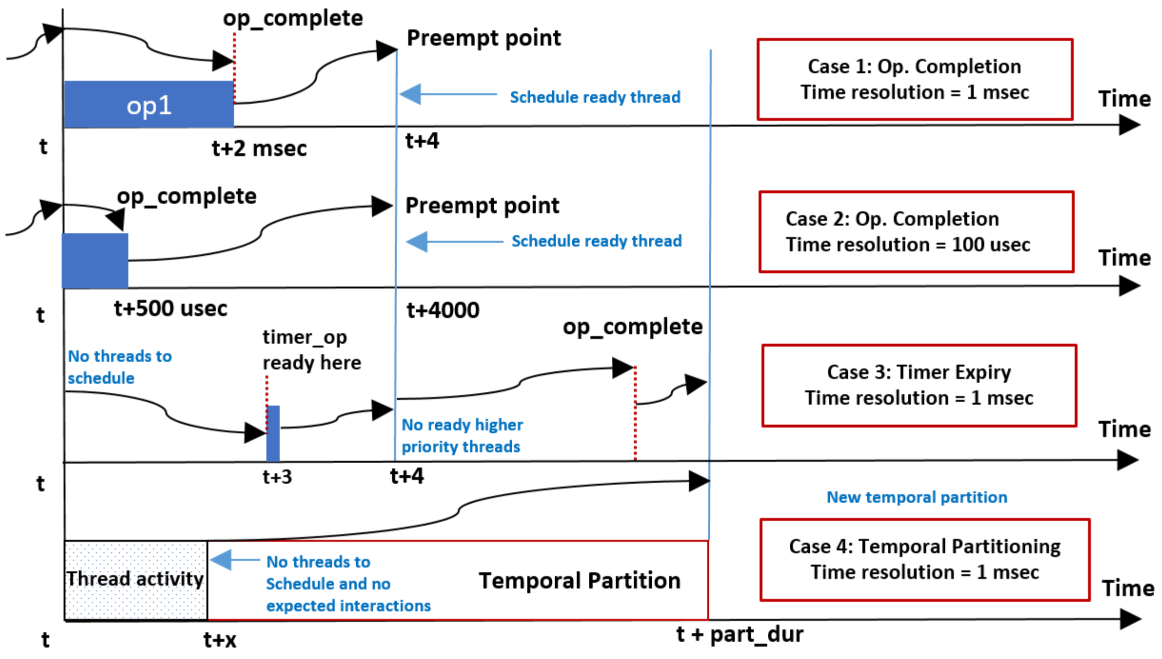
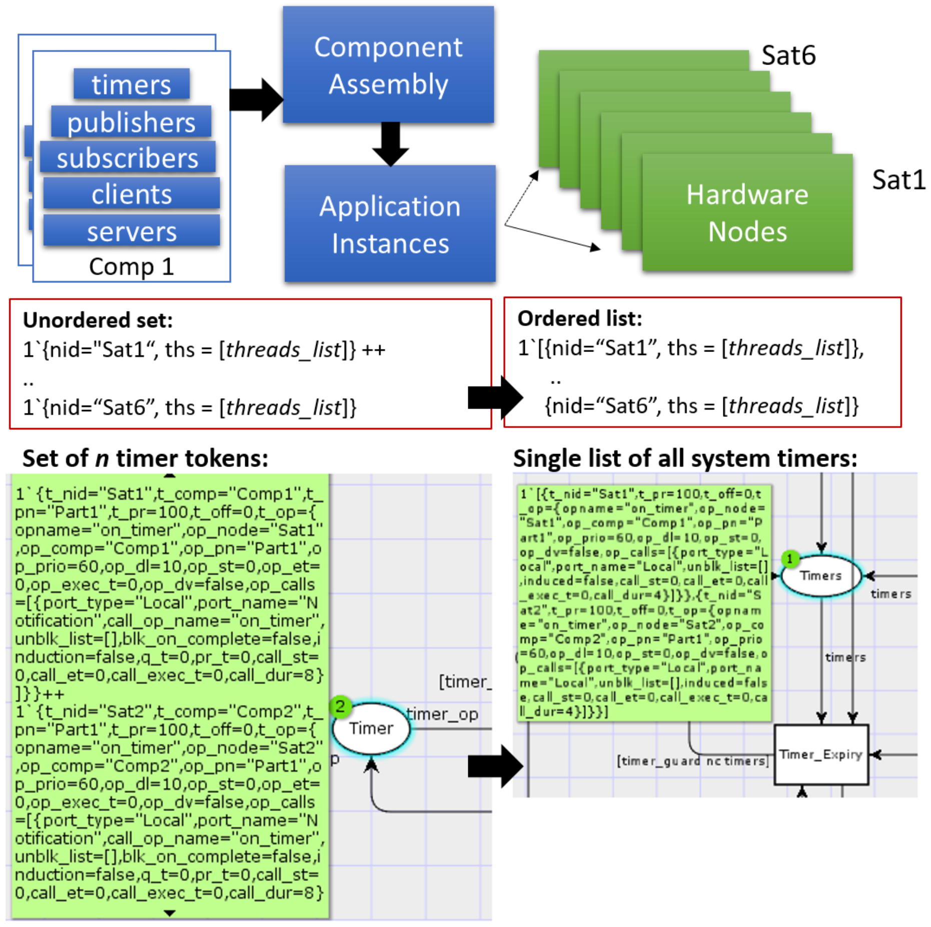

op_complete
Preempt point
op1
Schedule ready thread
Case 1: Op. Completion Time resolution = 1 msec
Time
t
t+2 msec
t+4
op_complete
Preempt point
Schedule ready thread
Case 2: Op. Completion Time resolution = 100 usec
Time
t
t+500 usec
t+4000
No threads to schedule
timer_op ready here
op_complete
No ready higher priority threads
Case 3: Timer Expiry Time resolution = 1 msec
Time
t
t+3
t+4
New temporal partition
Thread activity
No threads to Schedule and no expected interactions
Temporal Partition
Case 4: Temporal Partitioning Time resolution = 1 msec
Time
t
t+x
t + part_dur

Figure 21: Dynamic Time Progression

Figure 21 illustrates these time jumps using 4 scenarios. Assuming the scheduler clock ticks every 4 msec, Case 1 shows how time progression is handled when an operation completes 2 msec into its thread execution. At time t, the model identifies the duration of time left for an operation to complete. If this duration is earlier than the next preempt point, then there is no need to progress time in 1 msec increments as no thread can preempt this currently running thread till time t + 4 msec. Therefore, the *clock_value* in Figure 20 progresses to time t + 2 msec, where the model handles the implications of the completed operation. This includes possibly new interactions and operation requests triggered in other components. Then, time is forced to progress to the next preempt point where a new candidate thread is scheduled. This same scenario is illustrated in Case 2 when the time resolution is increased to 100 usec instead of 1 msec. Notice that the number of steps taken to reach the preempt point are the same, showing how the state space doesn't have to explode simply because the time resolution is increased. Case 3 illustrates the scenario where at time t, the scheduler has no ready threads to schedule since there are no pending operation requests but at time t + 3 msec, a component timer expires, triggering an operation into execution. Since timers are maintained in a global list, each time the *Progress_Time* transition checks its firing conditions, it checks all possible timers that can expiry before the next preempt point. So, at time t when no threads are scheduled, the model immediately jumps to time t + 3. This scenario also shows that if the triggered operation does not complete before the preempt point *and* there are no other ready threads or timer expiries that can be scheduled, the clock value jumps to the operation completion. It must be noted here that this case is valid only because the DREMS architecture we have considered uses a non-preemptive operation scheduling scheme. Lastly, Case 4 shows time jumps working with temporal partitioning. At some time t + x, the model realizes the absence of ready threads and does not foresee any interaction requests from other components, then it safely jumps to the end of the partition without stepping forward in 1 msec increments. This time progression directly shows how the state space of the system execution reduces while still preserving the expected execution order, justifying our choice of modeling time as a colored token using CPN.

3.7.2 Distributed Deployment

The second structural change to the analysis model is in how distributed deployments are modeled and simulated. Early designs on modeling and analysis of distributed application deployments [47] included a unique token per CPN place for each hardware node in the scenario. Since the individual *node* tokens are independent and unordered, there is non-determinism in the transition bindings when choosing a hardware node to schedule threads in. For instance, if there are 2 hardware nodes in the deployment with ready threads on both nodes, then either node can be chosen first for scheduling threads leading to two possible variations of the model execution trace. Therefore the generated state space would exponentially grow for each new hardware node. In order to reduce this state space and improve the search efficiency, we have merged hardware node tokens into a single *list* of tokens instead of a unassociated grouping of individual node tokens. This approach is inspired by the symmetry method for state space reduction [43].



Unordered set:
1`{nid="Sat1", ths = [*threads_list*]} ++
..
1`{nid="Sat6", ths = [*threads_list*]}

Ordered list:
1`[{nid="Sat1", ths = [*threads_list*]},
..
{nid="Sat6", ths = [*threads_list*]}]

Set of *n* timer tokens:

```
1`{t_nid="Sat1",t_comp="Comp1",t_
pn="Part1",t_pr=100,t_off=0,t_op={
opname="on_timer",op_node="Sat1"
,op_comp="Comp1",op_pn="Part1",o
p_prio=60,op_dl=10,op_st=0,op_et=
0,op_exec_t=0,op_dv=false,op_calls
=[{port_type="Local",port_name="N
otification",call_op_name="on_timer",
unblk_list=[],blk_on_complete=false,i
nduction=false,q_t=0,pr_t=0,call_st=
0,call_et=0,call_exec_t=0,call_dur=8}
]}}++
1`{t_nid="Sat2",t_comp="Comp2",t_
pn="Part1",t_pr=100,t_off=0,t_op={
opname="on_timer",op_node="Sat2"
,op_comp="Comp2",op_pn="Part1",o
p_prio=60,op_dl=10,op_st=0,op_et=
0,op_exec_t=0,op_dv=false,op_calls
=[{port_type="Local",port_name="N
otification",call_op_name="on_timer",
unblk_list=[],blk_on_complete=false,i
nduction=false,q_t=0,pr_t=0,call_st=
0,call_et=0,call_exec_t=0,call_dur=8}
```

Single list of all system timers:

```
1`[{t_nid="Sat1",t_pr=100,t_off=0,t
_op={opname="on_timer",op_node=
"Sat1",op_comp="Comp1",op_pn="P
art1",op_prio=60,op_dl=10,op_st=0,
op_dv=false,op_calls=[{port_type="L
ocal",port_name="Local",unblk_list=[]
,induced=false,call_st=0,call_et=0,call
_exec_t=0,call_dur=4}]}},{t_nid="S
at2",t_pr=100,t_off=0,t_op={opnam
e="on_timer",op_node="Sat2",op_co
mp="Comp2",op_pn="Part1",op_prio
=60,op_dl=10,op_st=0,op_dv=false,o
p_calls=[{port_type="Local",port_na
me="Local",unblk_list=[],induced=fals
e,call_st=0,call_et=0,call_exec_t=0,ca
ll_dur=4}]}}]
```

Figure 22: Structural Reductions in CPN

Figure 22 illustrates this structural reduction. Consider a distributed deployment scenario with an instance of a DREMS application deployed on each hardware node, Sat1 through Sat6. Components *Comp1* and *Comp2* are triggered by timers, eventually leading to the execution of component operations (modeled as shown in Figure **??**). If all the timer tokens in the system were modeled individually, the transition *Timer_Expiry* would non-deterministically choose one of the two timer tokens that are ready to expire at *t=0*. However, if the timers are maintained as a single list, then this transition (1) consumes the entire list, (2) identifies all timers that are ready to expire, (3) evaluates the timer expiration function on all ready timers, (4) propagates the output *operation* tokens to the relevant component message queues in a single firing. This greatly reduces the tree of possible transition firings and therefore the resultant state space. Also, if there is no non-determinism in the entire system, i.e., there is a distinct ordering of thread execution, then this model can be scaled up with instantiating the application on new hardware nodes with no increase in state space size. This is because all of the relevant tokens on all nodes are maintained as a single list that is completely handled by a single transition firing.

An important implication of the above structural reduction is that the simulation of the entire system now progresses in synchronous steps. This means that at time 0, all the timers in all hardware nodes that are ready to expire will expire in a single step. Following this, all operations in all component message queues of all these nodes are evaluated together and appropriate component executor threads are scheduled together. When these threads execute, time progresses as described in Section 3.7.1, moving forward by the minimum amount of time that can be fast-forwarded.

3.8 Investigating Advanced State Space Analysis Methods: COMPLETED

State space analysis techniques have been successfully applied with Colored Petri Nets in a variety of practical scenarios and industrial use cases [36], [38]. 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.

The variety of CPN-specific state space reduction techniques [16], [35] 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 [75] analysis tool. The tool provides for several search algorithms and state space reduction techniques such as the *sweep-line method* [17] which deletes already visited state space nodes from memory, forcing on-the-fly verification of temporal properties. The main advantage of such a technique 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 [17] 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 of similar reduction methods such as bit-state hashing [33] is that a complete state space search is guaranteed.

In order to illustrate the utility of such state space reduction techniques, we consider a large-scale deployment. Figure 23 shows the generated CPN model for a domain-specific DREMS application. This is a scaled-up variant of several satellite cluster examples we have used in previous publications [23, 47]. The example consists of a group of communicating satellites hosting DREMS applications. The component assembly for this application consists of 100 interacting components distributed across 10 computing nodes, many of which are triggered by infrastructural timers. Notice in Figure 23 how there is only one token in each of the main CPN places, as described in Section 3.7.2. All of the component timers are appended to the list maintained in *Timers* place. Similarly, all node-specific clock tokens are maintained in place *Clocks*.

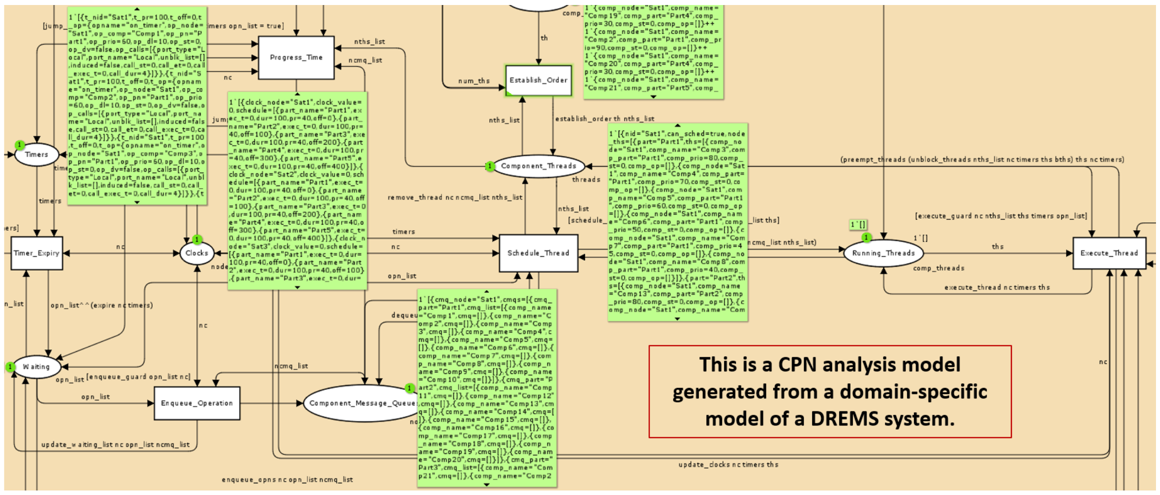
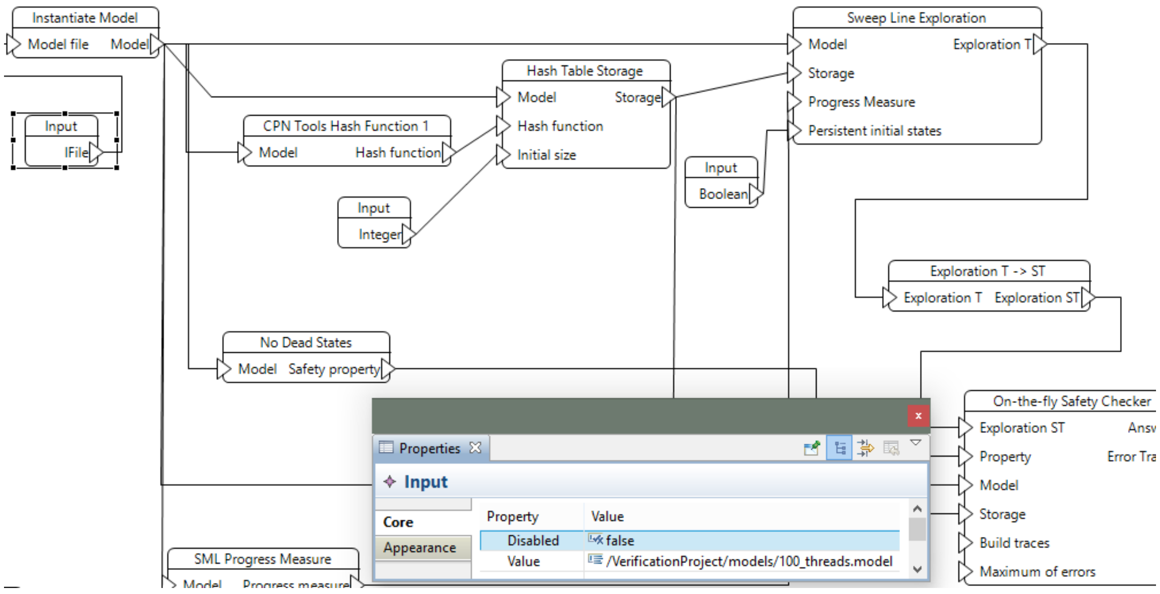


Figure 23: Generated CPN model for a Distributed Application Deployment

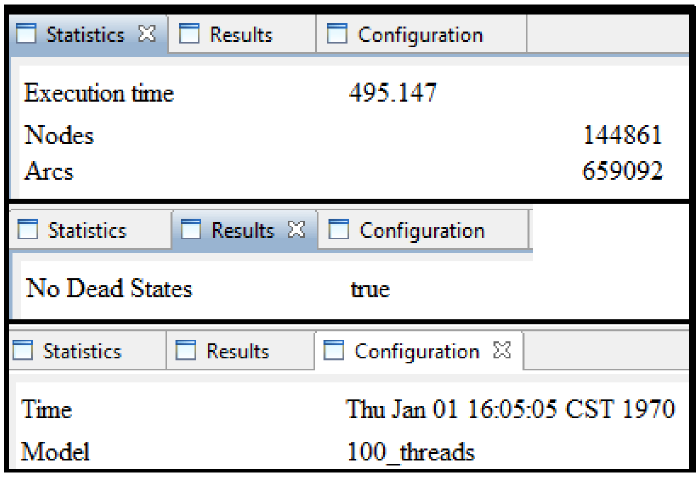
At time *t=0*, before the simulation is kicked off, the transition *Establish_Order* generates the powerset of thread execution orders that are possible given the configuration of the clock token. This may be a potentially large set depending on the number of threads of equal priority in each partition. Once this tree of possible orders is established, the complete set of timers that are ready to expire are evaluated. Each timer expiry manifests as an operation request and each callback operation modeled using the grammar shown in Figure 15. Once the operations are ready to execute, the highest priority component thread with a pending operation request is chosen for execution. This thread scheduling happens on all hardware nodes. When each thread executes, new interactions may occur as a consequence of the execution. For instance, if a component thread executes a timer operation in which the component publishes on a global topic, the consequence of this action would include a set of callback operation requests on all components that contain subscribers to that global topic. Lastly, all running threads are evaluated to identify the minimum amount of time that can be safely fast-forwarded in each node. If the running component threads are independent or symmetrical, then the maximum possible time progression is up to the end of the temporal partition. Note here that temporal partition in the deployment can be set to an empty list which simply removes the partitioning constraint and treats all component threads on a node as candidate threads for execution. The above sequence of transitions repeat for as long as there is a timer expiry, a pending operation request or an unfinished component interaction.



Property	Value
Disabled	false
Value	/VerificationProject/models/100_threads.model

Figure 24: Sweep-Line Method

Using the CPN Tools' built-in state space analysis tool, a bounded state space was generated reaching up-to 20 hyperperiods of component thread activity. This bounded generation took 36 minutes on a typical laptop. Our goal with such an example is to evaluate the effectiveness and utility of state space reduction techniques with respect to speed and memory usage. Figure 24 shows a simple block diagram of the sweep-line method as configured in ASAP. Performing on-the-fly verification checks for lack of dead states in the analysis model, results indicate lack of system-wide deadlocks due to blocking behaviors triggered by RMI-style synchronous peer-to-peer interaction patterns. Figure 25 shows analysis results obtained from a *Verification Job* executed in the tool. Notice the on-the-fly verification taking less than 10 minutes to perform deadlock checks on this sample deployment. Using the *Palette* in ASAP, several standard ML (SML) user queries can be created to check for domain-specific properties.



Execution time	495.147
Nodes	144861
Arcs	659092
No Dead States	true
Time	Thu Jan 01 16:05:05 CST 1970
Model	100_threads

Figure 25: Dead States Checking in a Component-based application

It must be noted here that this improved result is due to not only because of the efficient state space search but also because of symmetry-based structural reduction discussed in Section 3.7.2. If not for this reduction, the state space search requirements would exponentially grow for each new hardware node added to the deployment. These results were published in [46].

3.9 Experimental Validation of Timing Analysis Results: ONGOING

3.9.1 Challenges

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 [44] to simulate and analyze resilient cyber-physical systems. This testbed consists of 32 Beaglebone black development boards [1], ready for distributed real-time deployments. We have chosen the light-weight ROS [62] middleware layer and implemented our ROSMOD Component model [45] on top of it. This component model provides the same execution semantics and interaction patterns as our DREMS component model [56].

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 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.

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.

3.9.2 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.
- 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.

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

3.10.1 Challenges

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. The challenge here 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 [44] 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.

3.10.2 Proposed Contributions

- Model CPS I/O devices and interaction patterns between I/O devices and software components.
- Evaluate the modeling and analysis results by comparing against prototype CPS deployments on our testbed.
- Compare real-world execution logs with CPN analysis trace logs.

CHAPTER IV

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 2 provides the tentative timetable for the proposed research.

Table 2: Proposed Research Timetable

Thorough Experimental Validation	09/2015 - 11/2015
Modeling I/O components and analyzing CPS	10/2015 - 01/2016
Dissertation Writing	10/2015 - 03/2016

CHAPTER V

PUBLICATIONS

5.1 Highly Selective Publications

- 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

5.2 Other Conference and Workshop 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

- 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

- 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

- 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

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

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

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

[13] 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.

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

[15] 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.

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

[17] 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.

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

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

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

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

[28] 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.

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

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

[31] 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.

[32] 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.

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

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

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

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

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

[38] 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.

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

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

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. Ocarina: An environment for aadl models analysis and automatic code generation for high integrity applications. In *Reliable Software Technologies–Ada-Europe 2009*, pages 237–250. Springer, 2009.

[50] 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.

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

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

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

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

[59] 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.

[60] 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.

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

[62] 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.

[63] 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.

[64] 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.

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

[66] 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.

[67] 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.

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

[69] 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.

[70] 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.

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

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

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

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

[75] 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.

[76] 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.

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

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

[79] 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.

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

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

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