A Timing Analysis for Component-based Distributed Real-time Systems

Pranav Srinivas Kumar, Abhishek Dubey, and Gabor Karsai
ISIS, Dept of EECS, Vanderbilt University, Nashville, TN 37235, USA
Email:{pkumar, dabhishe, gabor}@isis.vanderbilt.edu

***Abstract*—This paper presents a timing analysis approach for modeling and verifying component-based software applications hosted on distributed real-time embedded (DRE) systems. Although schedulability analysis for real-time systems is a considerably well-studied field, various general-purpose timing analysis tools are not intuitively applicable to all system designs, especially when domain-specific properties such as hierarchical scheduling schemes, time-varying networks, and component-based interaction patterns directly influence the temporal behavior. Thus, there is still a need to develop analysis tools that are tightly coupled with the target system paradigm and platform while being generic and extensible enough to be easily modified for a range of systems. In this context, we have developed a Colored Petri Net-based schedulability analysis tool that integrates with a domain-specific modeling language and component model and that simulates and verifies temporal behavior for component operations in mission-critical DRE systems. Our results show the scalable utility of this approach for preemptive and non-preemptive hierarchical scheduling schemes in distributed scenarios.**

***Index Terms*—component-based, real-time, distributed, colored petri nets, timing, schedulability, analysis**

I. Introduction

Real-time systems are characterized by operational deadlines. These constrain the amount of time permitted to elapse between a stimulus provided to the system and a response generated by the system. Delayed response times and missed deadlines can have a catastrophic effect on the function of the system, especially in the case of safety and mission-critical applications. This is the primary motivation for design-time schedulability analysis and verification.

There is a wealth of existing literature studying real-time task scheduling theory and timing analysis in uniprocessor and multiprocessor systems [**?**], [**?**]. There are also several modeling, schedulability analysis and simulation tools [**?**], [**?**], [**?**], [**?**] that address heterogeneous challenges in verifying real-time requirements although many such tools are appropriate only for certain task models, interaction patterns, scheduling schemes, or analysis requirements. For component-based architectures, model-based system designs are usually transformed into a formal domain such as timed automata [**?**], [**?**], controller automata [**?**], high-level Petri nets [**?**] etc. so that existing analysis tools such as UPPAAL [**?**] or CPN Tools [**?**] can be used to verify either the entire system or its compositional parts. But, it is also evident that many of the existing schedulability analysis tools, though grounded in theory are not directly applicable to all system designs, especially with respect to domain-specific properties such as component interaction patterns, distributed deployment, time-varying communication networks etc.

To be useful, the analysis tools need to be tightly integrated with the target domain by which we mean here a model of computation. However, redeveloping analysis tools for every target domain is not desirable - we need to develop tools that can be applied to different domains with minimal modifications. For instance, the target domain in this paper is DREMS [**?**] which is a software infrastructure addressing challenges in the design, development and deployment of component-based flight software for fractionated spacecraft. The physical nature of such systems require strict, accurate and pessimistic timing analysis at design-time to avoid catastrophic situations. DREMS includes a design-time tool suite and a run-time platform built upon a Linux-based operating system that is enhanced by a well-defined component model. Domain-specific properties include a temporally and spatially partitioned scheduling scheme, a non-preemptive component-level operations scheduler, and semantically diverse interaction patterns in a distributed deployment that need to satisfy real-time requirements. While the analysis tool presented below reflects these features, it can be adapted to other target domains as well.

Our contributions in this paper target timing analysis of component-based applications that form distributed real-time embedded systems, such as in DREMS.

1) We present a scalable, extensible Colored Petri net-based [**?**] approach to abstracting the structural and temporal properties of model-based designs, including aspects such as component assembly, preemptive, and non-preemptive hierarchical scheduling, network-level delays, worst-case execution times, etc.
2) We identify the challenges faced in state space exploration and the heuristics used to efficiently verify the safe temporal behavior for large-scale distributed systems. Preliminary analysis results with these methods have been identified in [**?**].
3) Lastly, we briefly describe a simple language that explicitly models the component-level temporal behavior and enables model transformations and automated analysis.

The rest of this paper is organized as follows. Section **??** presents related research, reviewing and comparing existing analysis tools and formal methods. Sections **??** briefly describes the DREMS architecture, specifically the concepts of interest for timing analysis. Section **??** motivates the analysis approach by elaborating on implicit design challenges. Sections **??** and **??** show how DREMS concepts pertaining to a sample application are mapped into a modular, hierarchical CPN which is then analyzed. Section **??** describes integration of this analysis with design-time modeling tools. Finally, Sections **??** and **??** present possible future avenues of development for the analysis method and concluding remarks respectively.

II. Related Research

High-level Petri nets are a powerful modeling formalism for concurrent systems and have been integrated into many modeling tool suites for design-time verification. General-purpose AADL models have been translated into Symmetric nets for qualitative analysis [**?**] and Timed Petri nets [**?**] to check real-time properties such as deadline misses, buffer overflows etc. Similar to [**?**], our CPN-based analysis also uses bounded observer places [**?**] that observe the system behavior for property violations and prompt completion of operations. However, [**?**] only considers periodic threads in systems that are not preemptive. Our analysis is aimed at a combination of preemptive and non-preemptive hierarchical scheduling with higher-level component interaction concepts.

Verification of component-based systems require significant information about the application assembly, interaction semantics, and real-time properties. This information is primarily derived from the design model although many real-time metrics are not explicitly modeled. Using model descriptors, [**?**] describes interaction semantics and real-time properties of components. Using the MAST modeling and analysis framework [**?**], [**?**], schedulability analysis and priority assignment automation is supported. Event-driven models are separated into several *views* which are similar to hierarchical pages in CPN. Analysis efforts include the calculation of response times, blocking times and slack times.

Several analysis approaches present tool-aided methodologies that exploit the capabilities of existing analysis and verification techniques. In the verification of timing requirements for composed systems, [**?**] uses the OMG UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) modeling standard and converts high-level design into MAST output models for concrete schedulability analysis. In a similar effort, AADL models are translated into real-time process algebra [**?**] reducing schedulability analysis into a deadlock detection problem searching through state spaces and providing failure scenarios as counterexamples. Symbolic schedulability analysis has been performed by translating the task sets into a network timed automata, describing task arrival patterns and various scheduling policies. TIMES [**?**] calculates worst-case response times and scheduling policies by verifying timed automata with UPPAAL [**?**] model checking.

In order to analyze hierarchical component-based systems, the real-time resource requirements of higher-level components need to be abstracted into a form that enables scalable schedulability analysis. The authors in [**?**] present an algorithm where component interfaces abstract the minimum resource requirements of the underlying components, in the form of periodic resource models. Using a single composed interface for the entire system, the component at the higher level selects a value for operational period that minimizes the resource demands of the system. Such refinement is geared towards minimum waste of system resources.

III. Target System Architecture

The target architecture for timing analysis is *DREMS* [**?**], [**?**]. DREMS is a software infrastructure for the design, implementation, deployment, and management of component-based real-time embedded systems. The infrastructure includes (1) design-time modeling tools [**?**] that integrate with a well-defined (2) component model [**?**], [**?**] used to build component-based applications. Rapid prototyping and code generation features coupled with a modular run-time platform automate tedious aspects of software development and enable robust deployment and operation of mixed-criticality distributed applications. The formal modeling and analysis method presented in this paper focuses on applications built using this foundational architecture. We emphasize here the importance of a component model: it defines a model of computation with a strict component execution semantics.

A. Component Operations Scheduler

DREMS applications are built by assembling and composing re-useable units of functionality called *Components*. Each component is characterized by a (1) set of communication ports, a (2) set of interfaces (accessed through ports), a (3) message queue, (4) timers and state variables. Components interact via publish/subscribe and synchronous or asynchronous remote method invocation (RMI and AMI) patterns. Each component interface exposes one or more *operations* that can be invoked. Every operation request coming from an external entity reaches the component through its message queue. This is a priority-based queue maintained by a 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's message queue. The operation runs to completion, hence component execution is always single-threaded. Note however that the multiple components *can* be executed concurrently.

B. Operating System Scheduler

DREMS components are grouped into processes that are assigned to ARINC-653 [**?**] 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 *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.

C. Colored Petri Nets

Petri nets [**?**] 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, M_0) where P is a finite set of places, T is a finite set of transitions, A is a finite set of arcs between places and transitions, W is a function assigning weights to arcs, and M_0 is the initial marking of the net. Places hold a discrete number of markings called tokens. Tokens often represent resources in the modeled system. A transition can legally fire when all of its input places have necessary number of tokens. With Colored Petri nets (CPN) [**?**], 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.

IV. Analysis Challenges

There are certain implicit design complexities in DREMS that motivate our modeling and analysis approach.

A. Operation Deadlines

For each component the executor thread, when released by the component-level scheduler, runs under the control of the (fine-grain) OS scheduler. Each component executor thread has a fixed priority, assigned at design-time. However, the deadline for it is determined by the operation it executes, maintained at a higher level of abstraction. Therefore, depending on the operation executed by the executor thread, its timing requirements vary.

B. WCET of Operations

The execution of component operations serve the various periodic or aperiodic interaction requests from either the underlying framework 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 tasks, 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 dictates the WCET of the component operation. This behavior includes non-deterministic delays that are caused by components communicating through a (1) time-varying network (2) using specific interaction patterns and (3) within the constraints of temporally partitioned execution.

V. Colored Petri Net-based Analysis Model

This section briefly describes how Colored Petri nets (CPN) 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 [**?**] tool suite. The tool suite includes a simulator, as well as a state-space analysis tool that computes the (bounded) state space of the system under execution. The analysis model is based on the formal component model, i.e. the model of computation used. For a different component execution semantics, this model needs to be adjusted.

A. Model of Time

Appropriate choice for temporal resolution is a necessary first step in order to model and analyze threads running on a processor. The OS scheduler enforces temporal partitioning and uses a priority-based scheme for threads active within a temporal partition. If multiple threads have the same priority, a round-robin (RR) scheduling is used. In order to observe and analyze this behavior, we have chosen the temporal resolution to be 1 ms; a fraction of 1 clock tick of the OS scheduling quantum.

1) Dynamic Time Progression: Although the time resolution at the start of the analysis is chosen to be 1 ms, this is not a constant throughout the execution of the analysis model. If it were then 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, the analysis engine dynamically increases the time-step size and progresses time either to (1) the next node-specific clock tick, (2) the next global timer expiry offset, or (3) the next 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. Such dynamic control of time using global variables during analysis is also one of the advantages of using colored Petri nets.

B. Modeling Component-based Applications

The top-level CPN Model is shown in Figure **??**. 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 subnets to handle the hierarchical scheduling.

1) Component-level Execution Model:

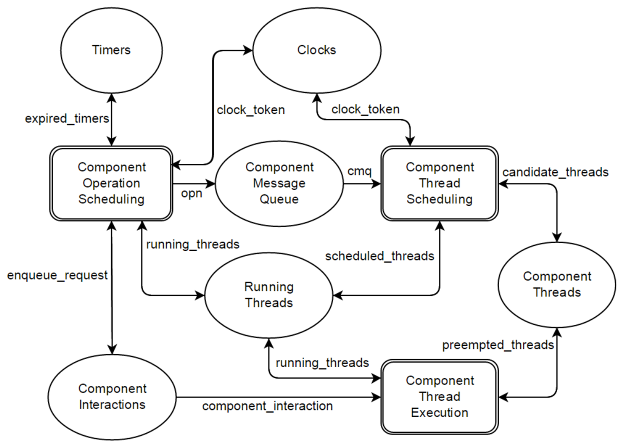


Fig. 1: Top-level CPN Model

a) 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.

b) 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 $0 \leq Exec_t \leq Dur_t$.

c) 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 **??**) and modeled as shown in equation **??**. 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.

d) 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. **??**. Timers are characterized by a period (Prd_{TMR}) and an offset (Off_{TMR}). Every timers triggers a component using the operation request O_{TMR}.

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

2) OS-level Execution Model:

a) Temporal Partitioning: The place *Clocks* in Figure **??** 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. **??**) 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 $Exec_{TP}$.

b) Component Thread Behavior: Figure **??** 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.

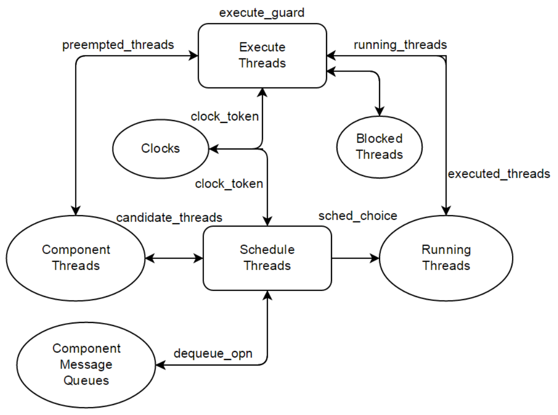


Fig. 2: Component Thread Execution Cycle

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.

VI. State Space Analysis

A. Sample Deployment

We briefly illustrate the timing analysis methods and lessons learned using a simple example. This is a 3-satellite cluster (Figure **??**) with each satellite running instances of a mixed-criticality DREMS application. Each satellite consists of a set of sensor-dependent critical components required for both safe flight and remote sensing tasks. These components interact with physical sensor devices such as cameras, GPS receivers, etc. periodically invoking functions, processing data and communicating with a ground station.

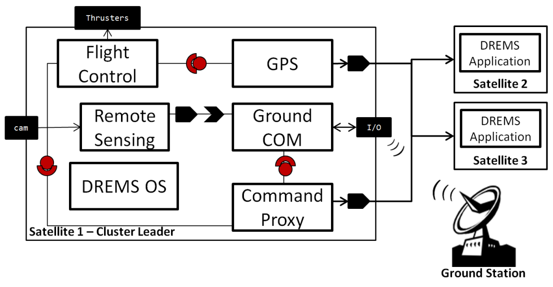


Fig. 3: A DREMS Application

A *GPS* component periodically publishes a state vector to all satellites in the cluster. The *Flight Control* component uses an RMI interface to access the most recent state vector from the GPS including data received from other satellites to calculate and maintain the flight trajectory. There are situations when the ground station will command the satellites in the cluster to perform a coordinated *scatter* maneuver; this is a time-critical event that is initiated by a transmission to the cluster leader satellite. On receiving such a command, the *Ground COM* of Satellite 1 uses an interface on the *Command Proxy* to publish this command to all satellites in the cluster. This proxy uses a high-priority synchronous RMI to communicate with the Flight Control component to trigger the scatter. Since the component-level scheduler is non-preemptive, the Flight Control cannot suspend any operation that it could be executing at time t_S when a scatter request is received from the Command Proxy. The abstract business logic *steps* of the *scatter* operation in the Flight Control is modeled as shown in Figure **??**.

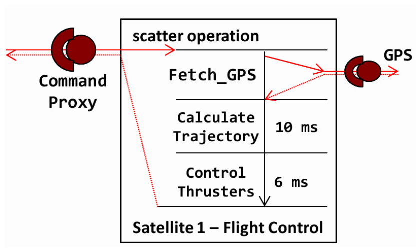


Fig. 4: Steps of the Scatter Operation

The temporal partition schedule for this scenario consists of two minor frames of length 100 msec each. All satellite sensor components such as the camera and the *GPS* are grouped into sensor processes and assigned to the first temporal partition. The *Flight Control* is assigned to partition 2 and is guaranteed to receive the newest vector data as it waits for the *GPS* component to update the database of state variables. The *Ground COM* and the *Command Proxy* component threads run at high priority and are scheduled on demand.

Although in a realistic deployment the network performance is dependent on the satellite's momentary position, we have assumed worst-case network delays on all data links. This is a simplification on the amount of information the CPN model needs to hold that can tightened by a more detailed, time-dependent model.

B. State Space Exploration

The CPN Tools uses a built-in state space (SS) analysis tool to generate a bounded state space from an initial CPN model. However, it has been previously noted [**?**] that the built-in state space tool is inefficient with its search algorithm. We therefore also use the ASAP tool [**?**] to utilize advanced analysis methods [**?**]. For smaller design models such as the above deployment, we also use observer places [**?**] in the CPN to *collect* tokens that represent timing anomalies such as deadline violations.

TABLE I: Component Operations on Satellite 1

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

Components in safety-critical DRE applications can be either sporadically or periodically triggered. A bounded state space generated in CPN Tools must be sufficiently complete so that the lack of deadline violations, deadlocks, or delayed responses *within* a chosen execution interval will guarantee safe operation throughout the complete lifetime of the applications. Table **??** shows some worst-case execution time results from state space analysis on each component thread on Satellite 1. Each operation request is enqueued into the component message queue at time T_{NQ}. A dispatcher thread dequeues this request at T_{DQ} and schedules an operation for execution. Once scheduled, the component executor thread executes each operation to completion at time T_{FIN} leading to an overall execution time of T_{EXEC} measured from T_{NQ}. These results allow the verification of several timing properties.

Lack of System-wide Deadlocks: System-wide deadlocks are caused by the inability of the OS schedulers (on any of the nodes) to schedule any component thread, e.g., due to infinite thread blocking. Deadlocks are identified by checking the firing conditions on leaf node transitions in the state space tree. Deadlock faults are also isolated by detecting operation timing violations, prolonged thread blocking (as observed in the place *Blocked Threads*) and idle scheduler behavior.

Lack Deadline Violations: A deadline violation is observed when a component operation takes longer to complete than its set *deadline*. The deadline is computed starting from the time when the operation is marked as *ready*. For every operation O that is either waiting in the message queue or currently executing, a violation *DLV(O)* is detected in accordance with Equation **??**.

$$DLV(O) = \begin{cases} true, & \text{if } T_{NC} - T_{NQ_O} > T_{DL_O} \\ false, & \text{otherwise} \end{cases} \tag{8}$$

where T_{NC} is the current node clock value and T_{NQ_O} is the time value at which the operation is enqueued onto the component message queue. If the elapsed time is longer than the operation's deadline T_{Dl_O}, a violation flag is set. At each state space node, the *SearchNodes* function of CPN Tools is used with the *DLV(O)* predicate to identify violated deadlines. However, the *search area* for this function must be carefully chosen. If the search area is small, the analysis may become shortsighted and incorrect. If the search area is too large (the entire tree), the query would take a long time to execute. For our simple example we generated a state space of upto 200,000 nodes covering a sufficiently large 10 hyperperiods of thread activity and varying thread priorities on all hardware nodes.

Worst-case Response time Analysis: Although a complete lack of deadline violations on the operation-level is important for timeliness of a component, it is often the case in distributed systems that a developer is more interested in end-to-end latencies between specific events. The response time for a system-level service may be dependent on timely execution of several *interacting* component operations. Specifying upper bounds for response times is also easier for a system integrator compared to providing deadlines for each operation. Here lies an analytical trade-off where a bounded trigger-to-response time would be sufficient even if one or more operations that enable the response violate individual deadlines.

The worst-case response time is computed as the difference between the earliest completion time of a triggering operation and the latest completion time of a response operation. Each of these values is derived from a bounded state space search query that identifies all *completed operations* and orders the resultant list based on T_{FIN}. For e.g. Table **??** shows a worst-case response time of 195 ms for the scatter command, observed when *GroundCOM* receives a return value from *scatter_cmd*. This response requires the timely execution of *notify_cmd*, *calc_trj* and *scatter* as also shown in Figure **??**.

1) Incomplete Designs: Initial designs of real-time systems are often specified by timing requirements between system entities. For scenarios where the developers are aware of minimum timing requirements but not thread execution orders or OS-level priorities, we have applied this approach to identify partial thread execution orders to refine incomplete designs. The results of this analysis help with identifying component threads that need to be separated by temporal partitions or thread priorities in order to satisfy timing guarantees. However, since all combinations of thread execution orders need to be observed to identify a useful *execution trace*, this particular analysis does not scale well for larger component-based applications where no priority assignment has previously been made.

C. Discussion

1) Conservative Results: Using estimates of worst-case execution time for component operations is motivated by the need to make exaggerated assumptions about the system behavior. Pessimistic estimates are a necessary requirement when verifying safety-critical DRE systems. Schedulability analysis with such assumptions should provide strictly conservative results. This means that (1) if the analysis results show the possibility of a deadline violation but the deployed system does not, the obtained result is a conservative one as it assumes worst-case behavior, and (2) if the analysis results do not show any timing violations but the deployed system violates response time requirements, deadlines etc., then the analysis does not provide a conservative result and has failed to verify system behavior.

In order to guarantee conservative results, the analysis must include worst-case behaviors of all system-level threads that run at higher priority than component threads and are not necessarily modeled by the design-time tools. In our analysis, these threads are grouped into a set of critical processes with approximations made to simulate the behavior of system-level threads such as (1) globally periodic CPU utilization, (2) CPU utilization for some WCET per partition. The best approximation is chosen based on the expected behavior of such critical processes. Best-effort processes are ignored as they always run at a priority lower than the lowest-priority component thread.

2) Scalability: In previous efforts [**?**], we presented results showing our analysis model that it scales well for medium-sized applications, tested up to 100 mixed-criticality components distributed on up to 5 computing nodes. As the determinism in the initial design increases, the number of possible behaviors and therefore the size of the state space decreases. In essence, the effort required for the analysis to be useful to the designer is dependent heavily on the initial design itself. Increasing the number of equally prioritized components will exponentially increase thread scheduling opportunities and the size of the state space required to accumulate the tree of component behaviors.

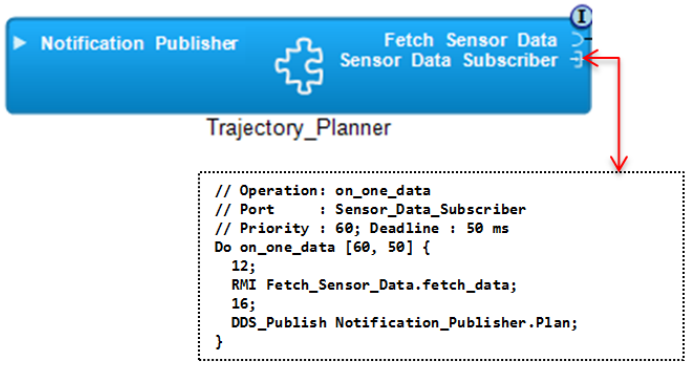
VII. Integration of Modeling Tools

This section briefly describes the integration of the CPN-based analysis with the design-time modeling tools for automated model transformations and code generation. By parsing the domain-specific model of the system, obtaining the necessary structural and behavioral attributes and using a model interpreter, a valid CPN model for the design is automatically generated.

A. Timing Specification for Component Operations

As mentioned in Section **??**, every component operation is modeled as a sequence of functional steps, each with an assigned worst-case execution time. Our goal with this specification is to be able to model these sequential steps. We broadly classified these steps into (1) local code blocks, (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 loops. The restriction to these 5 types is a consequence of the DREMS component model and can be easily expanded.

Figure **??** shows an example temporal specification for the operation *on_one_data* residing in Component *Trajectory_Planner* exposed through the *Sensor_Data_Subscriber* port. This port subscribes to sensor data published by another component using *Push Subscription* semantics. On reception of sensor data, the operation *on_one_data* is called, triggering a sequence of events. Abstracting the business logic, this operation has 4 execution steps: (1) local *work* for 12 ms, followed by an (2) RMI call to the remote method *fetch_data* using the *Fetch_Sensor_Data* receptacle port, (3) 16 ms of local work, and finally (4) publishing a *Plan* data structure anonymously using the *Notification_Publisher* port.



```
// Operation: on_one_data
// Port      : Sensor_Data_Subscriber
// Priority : 60; Deadline : 50 ms
Do on_one_data [60, 50] {
  12;
  RMI Fetch_Sensor_Data.fetch_data;
  16;
  DDS_Publish Notification_Publisher.Plan;
}
```

Fig. 5: Timing Specification for Component Operation

B. Model Generation

A model interpreter parses the design model tree and builds data structures representing the various CPN tokens described in Section **??**. This includes structural properties like component assembly and software deployment, temporal properties like scheduling schemes, operation execution times and instantiated deployment where a single application can be deployed simultaneously on multiple computing nodes. Using text templates, a modular and hierarchical CPN model is generated.

VIII. Future Work

The DREMS component communication is facilitated by a time-varying network. The bandwidth provided by the system therefore predictably fluctuates between a minimum and a maximum during the orbital period of the satellites. Currently, our analysis associates each operational step with a fixed worst-case network delay. We are working on introducing places in the CPN model that capture the *network profile* of a deployment so that the communication delays during queries vary with time leading to tighter bounds on predicted response times.

Secondly, we are also investigating the utility of this approach for fault-tolerant and self-adaptive systems. Integrating this analysis with a run-time resilience engine, configuration changes determined by a fault mitigating solver can be subsequently checked for timing anomalies before settling on a specific reconfiguration strategy.

IX. Conclusions

Mobile, distributed real-time systems operating in dynamic environments, and running mission-critical applications must satisfy strict timing requirements to operate safely. To reduce the development and integration complexity for such systems, component-based design models are being increasingly used. Appropriate analysis models are required to study the structural and behavioral complexity in such designs.

This paper presented a Colored Petri net-based approach to capture the architecture and temporal behavior of such component-based applications for both qualitative and quantitative schedulability analysis. The key idea behind the analysis is the precise modeling of the component execution semantics in the analysis tool. The analysis method is modular, extensible and intuitive and there are sufficient support tools to enable state space analysis and verification for medium to large size design models. We have investigated the challenges faced in preemptive and non-preemptive hierarchical scheduling schemes and the effect of component interaction patterns on real-time thread execution.

Acknowledgments: The DARPA System F6 Program and the National Science Foundation (CNS-1035655) supported this work. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of DARPA or NSF.