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 has been 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 responses 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 hierarchical scheduling, interacting components, distributed platforms with and time-varying communication networks etc.

There is still a need to develop analysis tools that are tightly coupled with the target system paradigm while being modular and generic enough that these tools can address the needs of a range of systems with minimal modifications. For instance, the target system 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 operation scheduling, and semantically diverse interaction patterns in a distributed deployment that need to satisfy real-time requirements.

Our primary contributions in this paper target timing analysis of component-based applications that form distributed real-time embedded systems, such as in DREMS. 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 preemptive and non-preemptive hierarchical scheduling, network-level delays etc. This also involves capturing the semantics and implementation-specific real-time properties of component interactions such as worst-case execution times, deadlines and blocking delays in the software design model. Such temporal specifications are estimated and modeled based on the target hardware platform and the communication network. We briefly describe a simple language that explicitly models this component-level temporal behavior and enables model transformations and automated analysis. Lastly, we identify the challenges of state space exploration and the heuristics used to efficiently verify the safe temporal behavior for large-scale distributed systems. Preliminary results of this work can be found in [**?**].

The rest of this paper is organized as follows. Section II presents related research, comparing with existing analysis tools and formal methods. Sections III briefly describes the DREMS architecture, specifically the concepts of interest for timing analysis. Section IV motivates the analysis approach by

elaborating on implicit design challenges. Sections V and VI show how DREMS concepts pertaining to a sample application are mapped into a modular, hierarchical CPN which is then analyzed. Section VII describes integration of this analysis with design-time modeling tools. Finally, Sections VIII and IX 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 estimates. 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 performed. 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.

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. Each component interface exposes one or more *operations* that can be invoked. Each operation contains *business logic* code written by a developer to implement its function. Every operation request coming from an external entity reaches the component through its message queue. This is a priority-FIFO queue maintained by a component-level scheduler that schedules operations for execution. When ready, a single *component executor thread* per component will be scheduled by the operating system 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.

B. Operating System Scheduler

DREMS components are grouped into processes that are assigned to temporal partitions, implemented by the DREMS OS scheduler. This scheduler was built by modifying the behavior of the standard Linux scheduler, introducing an ARINC-653 [**?**] style temporal and spatial partitioning scheme.

Temporal partitions are periodic, fixed intervals of the CPU's time. Threads associated with a partition are scheduled only when the partition is active. This enforces a temporal isolation between threads assigned to different partitions. The repeating partition windows are called *minor frames*. The aggregate of 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 dictates the amount of time available per hyperperiod to schedule all threads assigned to that partition.

IV. Motivation

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

A. Operation Deadlines

For each component, the OS scheduler schedules a component executor thread to execute operations scheduled by the component-level scheduler. Each component executor thread has a fixed-priority assigned at design-time. However, the deadline for a scheduled component thread is inherited from the operation it executes, maintained at a higher level of abstraction. Therefore, depending on the operation executed by a component 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. The business logic of 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 function call, initiate an interaction with another component, process a response from external entities, data-dependent control statements and loops etc. The behavior that is 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) with novel interaction patterns and (3) within the constraints of temporal partitioning.

System architectures like DREMS require an analysis tool with an expressive language that is tightly coupled with the target software platform (like our component model) to efficiently model and verify behavioral aspects of the system such as operational modes, the OS scheduling and distributed communication. Another crucial need for safety-critical systems is incorporating the timing analysis in every stage of the design. This firstly requires an analysis tool that can work with the non-determinism realized by incomplete designs. Secondly, this also requires the timing specifications and model transformations to be integrated into the design-time modeling. All of these aspects motivated us to choose colored Petri nets as the formalism to capture the various domain-specific properties and perform efficient timing analysis.

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 CPN-s 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.

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 lowest-level OS scheduler enforces temporal partitioning and uses a priority-based scheme for threads active within a partition. The highest priority thread is always chosen first for execution. This thread keeps hold of the CPU as long as it is runnable and there are no other threads of same priority that need to run. If multiple threads have the same priority, Round-Robin (RR) scheduling is enforced. Here, the scheduler allots a small quantum of time to one of the ready threads after which the thread is preempted by another ready thread of same priority. 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 system timer). Since the temporal resolution is set as a global integer variable in the model, this value can be raised or lowered depending on the nature of the system being analyzed.

1) Dynamic Time Progression: Although the time resolution at the start of the analysis is chosen to 1 ms, this is not a constant throughout the execution of the analysis model. If it was, then *S* seconds of activity will generate a state space of size:

$$SS_{size} = \sum_{i=1}^{S*1000} TF_{t_i} \tag{1}$$

where TF_{t_i} is the number of state-changing CPN transition firings after t_i and before t_{i+1}. This large state space includes periods of time where there might be no thread activity to analyze either due to lack of operation requests, lack of ready threads for scheduling or due to temporal partitioning. Therefore, during detected 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. Time jumps are also under effect for distributed cases when several computing nodes are simultaneously executing threads though it is enforced more carefully as to not lose any component interactions during the progressed time. This precise control of time 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 1. The places 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 enable modularity, this model is partitioned into two interacting sub-nets to handle the hierarchical scheduling.

1) Component-level Execution Model:

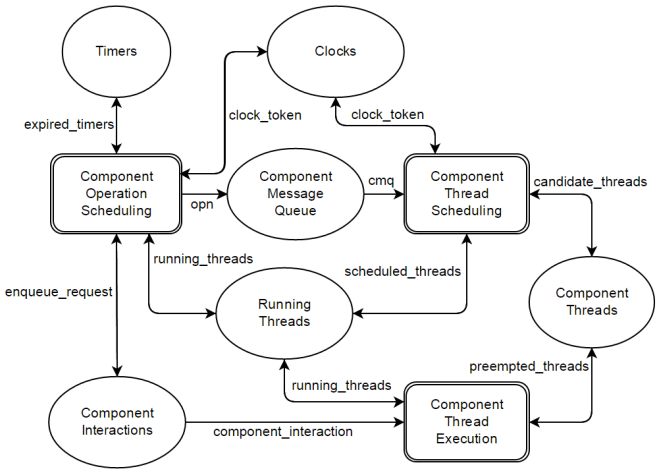

Fig. 1: Top-level CPN Model

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

$$O(C_x) = < ID_O,\ Prio_O,\ Dl_O,\ Steps_O > \qquad (2)$$

where, ID_O is a combination of strings that help identify and locate this operation in the system (name of (1) operation, (2) component, (3) computing node, (4) temporal partition etc.). 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 treated as a transition system that runs through a sequence of function calls dictating its behavior. Any of these underlying function calls can have a state-changing effect on the thread executing this operation. For example, RMI 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. Causes for such state changes propagating across the hierarchical boundary need to be captured within every component operation. 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:

$$s_i = < Port,\ Unblk_{s_i},\ Dur_t,\ Exec_t > \qquad (3)$$

where $1 \leq i \leq m$. *Port* is a *record* representing the exact communication port used by the operation during s_i. The list $Unblk_{s_i}$ is a list of component threads that are unblocked when s_i completes. This list is applicable, for example, when the completion of a synchronous remote method execution 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 state of the execution s_i where $0 \leq Exec_t \leq Dur_t$.

c) Component Interactions: To handle component interactions, we take advantage of the developer's knowledge of the structure of the application, i.e. the component wiring. Consider an application with two components, a client and a server. The client is periodically triggered by a timer to make an RMI call to the server. We know here that when the client executes an instance of the timer-triggered operation, a related RMI 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 RMI activity. Tokens that represent such design-specific interactions are maintained in *Component Interactions* (Figure 1) and modeled as shown in equation 4. The interaction *Int* observed when a component C_x queries another component C_y is modeled as the 3-tuple:

$$Int(C_x, C_y) = < Node_{C_x},\ Port_{C_x},\ O(C_y) > \qquad (4)$$

When an operational *step* in component C_x uses port $Port_{C_x}$ to query an operation on component C_y, the operation request O_{C_y} is enqueued on the message queue of C_y. As shown in Figure 1, the transition *Component Operation Scheduling* observes the *Running Threads* at every time step to check for completion of external interaction requests. When satisfied, the necessary operation token is removed from *Component Interactions* and enqueued onto the appropriate message queue.

d) Timers: DREMS components are dormant by nature. 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 setup to trigger a component. When the component's timer expires, a timer operation request is placed on the message queue. When dequeued, a timer callback is executed by the component. In CPN, each timer TMR is held by the place *Timers* and represented as shown in Eq. 5. Timers are characterized by a period (Prd_{TMR}) and an offset (Off_{TMR}). Every timers triggers a component using the operation request O_{TMR}.

$$TMR = < Prd_{TMR},\ Off_{TMR},\ O_{TMR} > \qquad (5)$$

2) OS-level Execution Model:

a) Temporal Partitioning: The place *Clocks* in Figure 1 holds the state of the node-specific global clocks. The state of the temporally partitioned schedule modeled within these clocks enforces a constraint on both the set of component operations and the set of component threads that can be scheduled. Each clock token *NC* is modeled as a 3-tuple:

$$NC = < Node_{NC},\ Value_{NC},\ TPS_{Node_{NC}} > \qquad (6)$$

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.

$$TP = < Name_{TP},\ Prd_{TP},\ Dur_{TP},\ Off_{TP},\ Exec_{TP} > \quad (7)$$

Each partition TP (Eq. 7) is modeled as a record color-set consisting of a name $Name_{TP}$, a period Prd_{TP}, a duration Dur_{TP}, an offset Off_{TP} and the state variable $Exec_{TP}$. Aggregate of such partitions can fully describe a partition schedule. Complete partition schedules are maintained per computing node.

b) Component Thread Behavior: Figure 2 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:

$$CT = < ID_{CT},\ Prio_{CT},\ O_{CT} > \quad (8)$$

where ID constitutes the set of strings required to identify a component thread in CPN, e.g. 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. The DREMS OS uses a fixed-priority preemptive scheduling scheme (constrained by temporal partitioning) to schedule component threads. If multiple threads belonging to the same temporal partition are ready, the highest priority thread is always chosen for execution. If there is more than one candidate thread, one is chosen at random, followed thereafter by a Round-Robin scheduling scheme.

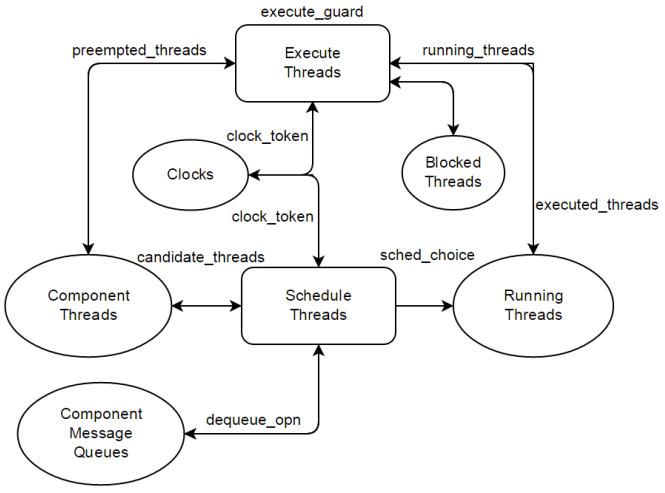

Fig. 2: Component Thread Execution Cycle

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*. The guard on *Schedule Thread* ensures that the highest priority thread in the currently running partition is scheduled first.

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 *Execute Thread* which also handles a part of the time progression. Keeping track of $Value_{NC}$, the thread is preempted at each clock tick. This loop repeats as long as there are no system-wide deadlocks.

It must be noted that system-critical processes such as deployment and fault management processes running periodically (or sporadically) at higher priorities than component processes are not necessarily affected by temporal partitioning. Such processes are scheduled to execute in a *SYSTEM* partition when ever ready. Therefore when simulating the scheduling of such process threads, $TPS_{Node_{NC}}$ in equation 6 is ignored.

VI. State Space Analysis

A. Experimental Deployment

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

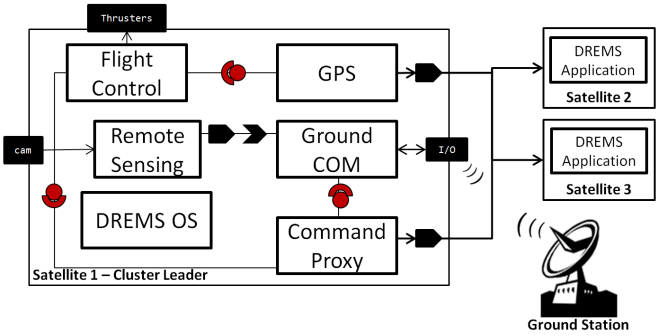

Fig. 3: DREMS Application

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

A more critical task in this deployment is maintaining the cluster attitude. This includes the relative position and acceleration of the satellites from each other with the satellites using a distributed but synchronized database to keep track of *state vectors* of each other. A *GPS* component periodically publishes a state vector to all satellites in the cluster. The GPS component uses its subscriber port to receive the equivalent vectors from other satellites. A *Flight Control* component responsible for the integrity of the cluster flight uses an RMI

interface to access the most recent vector metrics from GPS including metrics received from other satellites to calculate and maintain the flight trajectory. The Flight Control component does not fetch GPS information until all the satellites have published on the updated sensor data.

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

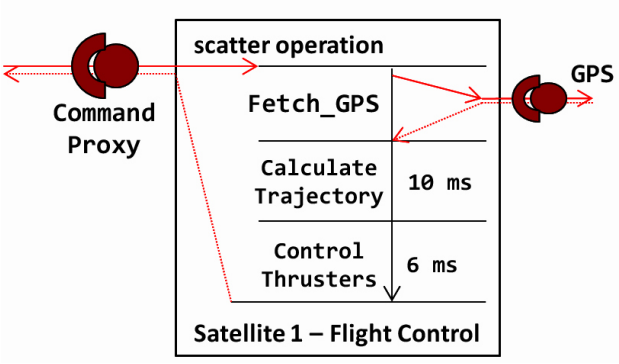

Fig. 4: Scatter Operation

The abstract business logic *steps* of the *scatter* operation in the Flight Control is modeled as shown in Figure 4. This operation is requested by the Command Proxy when a command is received from the ground station. For sake of simplicity, we have reduced this operation down to 3 distinct steps. The controller first obtains updated position vectors from the GPS component using the RMI interface. Notice the lack of a WCET for this interaction. This is because the GPS is scheduled on a separate temporal partition and the amount of time for which the Flight Control has to wait for the updated metrics is dependent on the time stamp at which the scatter command is received and also the state of the GPS component message queue. Once these metrics are received, the Flight Control calculates a new trajectory which is also the return arguments of this operation. This trajectory is passed down to the Command Proxy which publishes the leader's updated metrics to the rest of the satellites in the cluster. Once the trajectory is calculated, each Flight Control component interacts with the thrusters and maneuvers the satellites.

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

The Flight Control is assigned to partition 2 and is guaranteed to receive the newest vector data as it waits for the GPS component to update the database of state variables. The Ground COM and the Command Proxy component threads run at SYSTEM priority and are scheduled as and when required. In our case, we use a sporadic timer on the Ground COM component to trigger the sequence of interactions leading a scatter.

B. State Space Exploration

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

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

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

As mentioned earlier, our model uses an observer place for deadline violation detection, especially in small/medium sized applications. A *Deadline_Violation* (Figure 5) transition fires

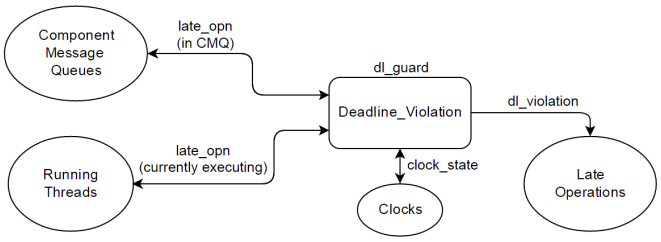

Fig. 5: Deadline Violation Observer place

at any point in time when the guard *dl_guard* is satisfied and arc bindings are realized with its input places. The transition observes the states of the currently running threads and the component message queues to identify deadline violations on operations that are either executing or waiting to execute. The *dl_violation* tokens in *Late Operations* (*LO*) is of the form:

$$LO = < Node_{name}, O_{name}, O_{ST},\ O_{DLT} > \qquad (9)$$

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

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

System-wide deadlocks are caused by the inability of the OS schedulers (on all nodes) to schedule any component thread. This can be caused by situations where a set of executing threads are indefinitely blocked on each other because of cyclic dependencies in the interactions. Deadlocks can identified by checking the leaf nodes of the bounded state space for *dead transitions* that are unable to fire. Alternatively, the tokens in *Component Interactions* are analyzed to identify cyclic dependencies and provide warnings to possible deadlocks. Such queries are useful in large component assemblies where mutually blocking dependencies are not immediately perceivable.

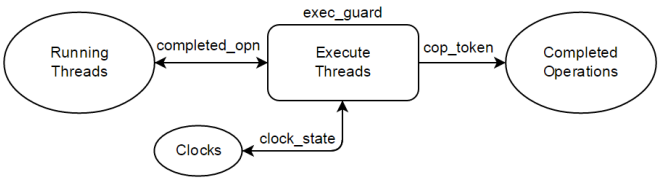

Fig. 6: Response-time Analysis

3) Response-time Analysis: Response-times are measured by observing completed operations. Similar to deadline violations, an observer place *Completed Operations* (Figure 6) accumulates all operation requests completed by component threads on all nodes. The structure of this data is similar to equation 9 except we replace O_{DLT} with the end-time of the operation O_{ET}.

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

4) Search Results: Table I shows some worst-case execution time results from state space analysis on each component thread on Satellite 1. Each operation request is enqueued into the component message queue at T_{NQ}. A dispatcher thread dequeues this request at T_{DQ} and schedules an operation for execution. Delays in the dequeue can be caused due to the non-preemptive nature of the component-level scheduling. Once scheduled, the component executor thread sequentially executes the steps in each operation to completion at time stamp T_{FIN} leading to an overall execution time of T_{EXEC} measured starting from T_{NQ}.

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

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

Consider a sample application that consists of 6 components servicing operation requests. Components threads 1, 2, and 3 are assigned to Partition 1 and threads 4, 5, and 6 are assigned to Partition 2. The thread priorities and execution orders are unknown. Each component is triggered by a timer once every major frame, taking up to 8 ms to complete the callback operation. Assuming a designer requires that operation 3 (handled by thread 3) must complete before 20 ms

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

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

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 strictly provide conservative results. This means that:

- If the analysis results show the possibility of a deadline violation but the deployed system does not, the obtained result is a conservative one as it assumes worst-case behavior.
- If the analysis results do not show any timing violations but the deployed system violates response time requirements, deadlines etc., then the analysis does not provide a conservative result and has failed to verify system behavior.

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

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

a) Symmetry: One of the main advantages of using component-based design for complex systems is reusability. It is not unusual to deploy instances of the same application with the same component assembly on multiple computing nodes. If these applications are deployed on symmetrical nodes, the observed state space of behaviors are also symmetrical. To enable efficient search through symmetrical state spaces, we use symmetry-based state space reduction techniques [**?**] to improve the performance. In Figure 1, to model 100 timers, we moved from using 100 colored tokens in the *Timers* place to using a single token which is a list consisting of a 100 elements. This way, when multiple timers expire, a single transition firing handles all the expiries across all nodes (a single state change) instead of multiple transition firings leading to multiple new states. This works well when a single instance of an application is deployed on several computing nodes with little non-determinism. The non-determinism caused by the OS-level scheduling of components and component-level operation request collisions still cause the state space to grow across symmetric nodes.

b) ASAP Tool: The CPN Tools GUI contributes to inefficient performance of state space generation for large CPN models. This is significantly improved by using the ASAP [**?**] analysis tool. The tool provides for several search algorithms and state space reduction techniques such as the *sweep-line method* [**?**] which deletes already visited state space nodes from memory, forcing on-the-fly verification of temporal properties. One of the disadvantages of using such heuristics is the run-time cost incurred by backward-generation of states when a backtrace for a violation is required. However, if a path

from the initial state to an inconsistent state is not desired, a combination of this method and the symmetry method reduces the state space and improves verification performance.

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 generation. For large applications with multiple computing nodes, application instances and component assemblies, hand-writing the CPN token specifications will prove to be cumbersome and error prone. This is avoided completely by parsing the domain-specific model of the system, obtaining necessary structural and behavioral attributes and using a model interpreter to generate a valid CPN model for the design.

A. Timing Specification for Component Operations

As mentioned in Section V-B1b, every component operation is modeled as a sequence of functional steps, each with an assigned worst-case execution time. When scheduled, these operations execute one step at a time. Our goal with this specification is to be able to model these sequential steps. We broadly classified these steps into (1) local uninvolved code blocks, (2) peer-to-peer synchronous and asynchronous remote calls, (3) anonymous publish/subscribe distribution services, (4) blocking and non-blocking I/O interactions and (5) control loops. The restriction to these 5 types is a consequence of the coupling with the DREMS component model and can be easily expanded.

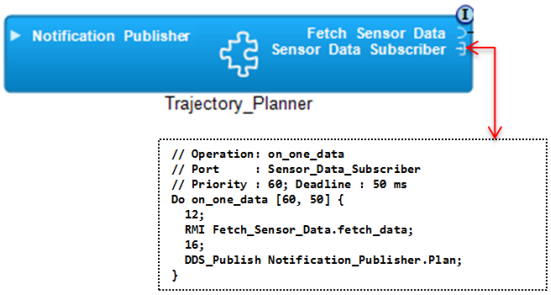

Fig. 7: Timing Specification for Component Operation

Figure 7 shows the 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 (3) 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.

B. Model Generation

CPN Tools saves colored Petri nets using an XML-based text template. The goal of model generation is to accurately generate a valid *.cpn* file from the domain-specific model of the component-based application. A model interpreter parses the design model diagrams and accumulates data structures representing the various CPN tokens described in Section V. This includes structural properties like component assembly and software deployment, temporal properties like scheduling schemes, operation execution times and instanced deployment where a single application can be deployed simultaneously on multiple computing nodes. Figure 8 shows the token structure generated for the on_one_data operation.

```
{opname="on_one_data", op_node="Sat1", op_comp="Trajectory_Planner",
    op_pn="Partition1", op_prio=60, op_dl=50,
op_steps=[{port_type="Local", port_name="Local",
                unblk_list=[], call_exec_t=0, call_dur=12},
          {port_type="RMI_Receptacle", port_name="Fetch_Sensor_Data",
                unblk_list=[], call_exec_t=0, call_dur=0},
          {port_type="Local", port_name="Local",
                unblk_list=[], call_exec_t=0, call_dur=16},
          {port_type="Publisher", port_name="Notification_Publisher",
                unblk_list=[], call_exec_t=0, call_dur=0]}}
```

Fig. 8: on_one_data CPN token

Using run-time text templates [**?**], a modular and hierarchical CPN model is generated. This CPN model is accompanied by a set of *.sml* function files that dictate the behavior of the CPN transitions, guards and arc bindings. Using parameters in the design model, these functions can be easily interchanged to tweak the behavior of the analysis model e.g. component-level EDF or FIFO scheduling, OS-level scheduling schemes and state space analysis options.

VIII. Future Work

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. Remote systems like DREMS require autonomous resilience and design-time verification of system configurations. Challenges in this domain include formal modeling and analysis of faults in the component behavior and timing verification of configuration points identified by some resilience engine to stabilize a faulty system.

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 presents 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 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.