Colored Petri Net-based Modeling and Formal Analysis of Component-based Applications

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*— Distributed Real-Time Embedded (DRE) Systems that address safety and mission-critical system requirements are applied in a variety of domains today. Complex, integrated systems like managed satellite clusters expose heterogeneous concerns such as strict timing requirements, complexity in system integration, deployment, and repair; and resilience to faults. Integrating appropriate modeling and analysis techniques into the design of such systems helps ensure predictable, dependable and safe operation upon deployment.**

In this paper, we present a Colored Petri net-based approach to modeling and analysis of component-based software applications for enabling the verification of system properties such as lack of deadline violations. The approach is based on (1) formalizing the component operation scheduling using Colored Petri nets (CPN), (2) modeling the abstract temporal behavior of application components, and (3) integrating the business logic and the component operation scheduling models into a concrete CPN, which is then analyzed. This model-driven approach enables a verification-driven workflow wherein the application model can be refined and restructured before actual code development. Thereafter, the abstract temporal behavior of the application components provide the necessary guidelines for the application developers to structure their code.

***Index Terms*—Colored Petri nets, modeling tools, design, analysis, verification, real-time, schedulability**

I. Introduction

Safety and mission-critical DRE systems are used in a variety of domains such as avionics, locomotive control, industrial and medical automation. Given the increasing role of software in such systems, growing both in size and complexity, utilizing predictable and dependable software is critical for system safety. To mitigate this complexity, model-driven, component-based software development has become an accepted practice. Applications are built by assembling together small, tested component building blocks that implement a set of services. Models describe what these component blocks are, what interfaces they have, how they are built, how they interact and how they are deployed to realize the domain-specific application.

Complex, managed systems, e.g. a fractionated spacecraft following a mission timeline and hosting distributed software applications expose heterogeneous concerns such as strict timing requirements, complexity in deployment, repair and integration; and resilience to faults. High-security and time-critical software applications hosted on such platforms run concurrently with all of the system-level mission management and failure recovery tasks that are periodically undertaken on the distributed nodes. Once deployed, it is often difficult to obtain low-level access to such remote systems for run-time debugging and evaluation. These types of systems therefore demand advanced design-time modeling and analysis methods to detect possible anomalies in system behavior, such as unacceptable response time, before deployment.

Our team has designed and prototyped a full information architecture called **D**istributed **RE**al-time **M**anaged **S**ystem (DREMS) [**?**], [**?**] that addresses requirements for rapid component-based application development. In prior work, we have described the design-time modeling capability [**?**], and the component model used to build and execute applications [**?**]. The formal modeling and analysis method presented in this paper focuses on applications that rely on this foundational architecture. The principle behind design-time analysis here is to map the structural and behavioral specifications of the system under analysis into a formal domain for which analysis tools exist. The key is to use an appropriate model-based abstraction such that the mapping from one domain to another remains valid under successive refinements in system development such as code generation. The analysis must ensure that as long as the assumptions made about the system hold, the behavior of the system lies within the safe regions of operation. The results of this analysis will enable system refinement and re-design if required, before actual code development.

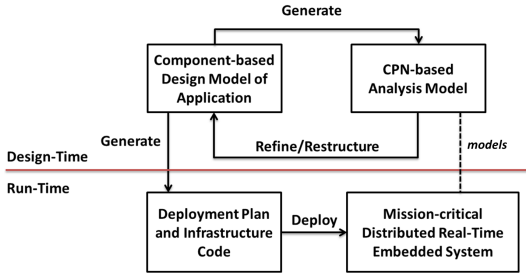

Fig. 1: Verification-driven Workflow

Figure 1 shows a Verification-driven workflow for component-based software development. Application developers use domain-specific modeling languages to model the component assembly, interaction patterns, component execution code, sequence of operations, and associated temporal properties such as estimated execution times, deadlines etc. Using such application-specific parameters in the *design* model, a Colored Petri net-based (CPN) [**?**] *analysis* model is generated. The system behavior is analyzed to check properties like lack of deadlocks, deadline violations etc.

The remainder of this paper is organized as follows. Section II presents existing research relating to this paper and explains how this approach is different; Section III provides a brief background on the DREMS Infrastructure and on the CPN formalism; Section IV discusses the problem statement that is evaluated; Section V elaborates on how this architecture is abstracted and modeled using CPN; Section VI investigates the utility and scalability of this approach in detecting deadline violations, calculating worst-case trigger-response times and determining partial thread execution orders; Section VII briefly describes how the analysis model can be generated; Sections VIII, IX and X present a brief discussion of the presented results, future extensions to the proposed approach and concluding remarks respectively.

II. Related Research

In recent years, much of the proliferating work in the development of mission-critical distributed real-time systems addresses the need for Safety and Verification-driven Engineering. Structural properties of the system are established using domain-specific modeling tools. Design models are transformed into relevant analysis models to study probable behaviors of the system and identify anomalies. When analyzing timing behavior, typically several exaggerated assumptions are made about the system behavior. These include upper bounds on task execution times, service rates, maximum resource consumption etc. The results of system analysis using such assumptions are equally pessimistic. However, real-time systems with high criticality necessitate such pessimistic assumptions to avoid the consequences of poor design. Predictability of system behavior is achieved by obtaining upper bounds of the system properties.

Petri nets and their extensions have proven to be a powerful formalism for modeling and analyzing concurrent systems. System designs represented using a domain-specific modeling languages are often translated into Petri nets for formal analysis. High-level formalisms such as AADL models have been translated into Symmetric nets for qualitative analysis [**?**] and Timed Petri nets [**?**] to check for real-time properties such as deadline misses, buffer overflows etc. Similar to [**?**], our CPN-based analysis also makes use of observer places [**?**] that monitor the system behavior and look for real-time property violations and prompt completion of operations. However, [**?**] only considers periodic threads in systems that are not preemptive. Our analysis covers a broader range of thread interaction patterns geared towards component-based applications operating on a hierarchical scheduling scheme requiring higher-level modeling concepts to capture component interaction in a distributed setup.

In the context of component-based systems, for complete real-time analysis, it is necessary to obtain significant information about the component assembly, the interaction patterns and real-time data about the temporal behavior of components. The real-time model of the system is composed of real-time models of its constituent parts, each with its own temporal behavior. Using abstract model descriptors, [**?**] describes a real-time model for component-based systems, including semantic and quantitative meta-data about component real-time behavior. Using the MAST transactional modeling methodology [**?**], [**?**] and analysis tools in the MAST environment, schedulability checks and priority assignment automation are performed. It must be noted here that for every real-time application, a separate and independent real-time analysis model is generated for each mode of operation and analyzed separately.

For classes of component-based systems whose component assembly and application structure change dynamically over time, design-time verification is observed to be insufficient. Incremental re-verification strategies [**?**] have been applied on dynamic systems to augment traditional compositional verification by identifying the minimal set of components that require re-verification after dynamic changes. Since our approach considers design-time deployment plans that are static, the analysis does not consider dynamic changes to component assembly at run-time.

III. Background

A. DREMS Infrastructure

1) DREMS Components: Design and implementation of component-based software applications rests on the principle of assembly: *Complex systems can be built by composing re-useable interacting components*. Components contain functional, business logic code that implements operations on state variables. Ports facilitate interactions between communicating components. A component-level message queue, with associated infrastructure code, controls the scheduling of operations on the individual components. Figure 2 shows a basic DREMS component.

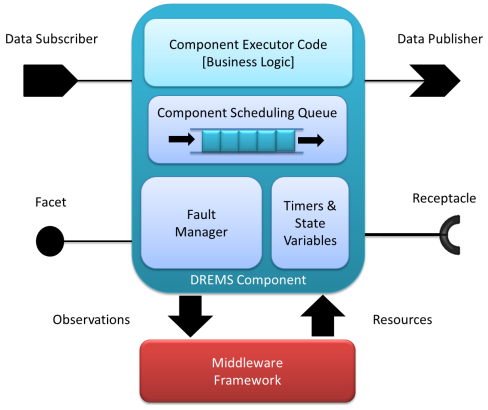

Fig. 2: DREMS Component

Each DREMS component supports four basic types of ports for interaction with other collaborating components: Facets, Receptacles, Publishers and Subscribers. A component's **facet** is a unique interface that it exposes to its clients. This interface can be invoked either synchronously via remote method invocation (RMI) or asynchronously via asynchronous method invocation (AMI). A component's **receptacle** specifies an interface required by the component in order to function correctly. Using its receptacle, a component can establish

connections and invoke operations on other components using either RMI or AMI. A **publisher** port is a single point of data emission. This port emits data produced by a component operation. A **subscriber** port is a single point of data consumption, feeding received data to the associated component. Communication between publishers and subscribers is contingent on the compatibility of their associated topics. Publishers and Subscribers enable the OMG DDS anonymous publish/subscribe style of messaging. More details on this component model can be found in [**?**].

2) Component Operations: An operation is an abstraction for the different tasks undertaken by a component. These tasks are implemented by the component's executor code written by the developer. In order to service interactions with the underlying framework and with other components, every component is associated with a message queue. This queue holds instances of operations ('messages') that are ready for execution and need to be serviced by the component. These operations service either interaction requests (seen on communication ports) or service requests (from the underlying framework). An example for the latter is the use of component timers that can periodically (or sporadically) activate an operation.

Figure 3 shows the basic structure of this model. Each operation is characterized by a priority and a deadline. Operation deadlines are quantified in absolute time measured starting from when the operation is enqueued onto the component message queue.

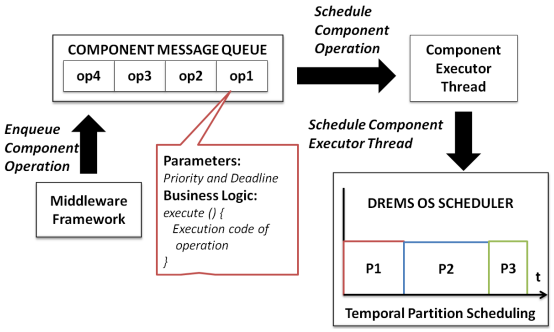

Fig. 3: Scheduling Component Operations

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

Figure 4 shows a sample timing diagram of how a component operation is handled. At time 0, the component executor thread is running some previously ready component operation, *op1*. At this point, consider that a new component operation *op2* is enqueued onto the message queue and marked as ready. At time 10, assuming that no other component requires scheduling, the component dispatcher thread of this component dequeues the next ready operation for execution. Assuming the component executor thread is scheduled immediately by the underlying processor, this thread runs the ready operation by invoking its *execute* function. This operation is run to completion at time 16. The total time taken for execution of this operation is measured from when the operation was enqueued, i.e. time 0. If the time taken for the operation exceeds its deadline, a fault manager is immediately notified. The duration of the component operation is further delayed by temporal partitioning enforced by the OS scheduler. This adds to the need for schedulability analysis, especially in case of safety and mission-critical applications.

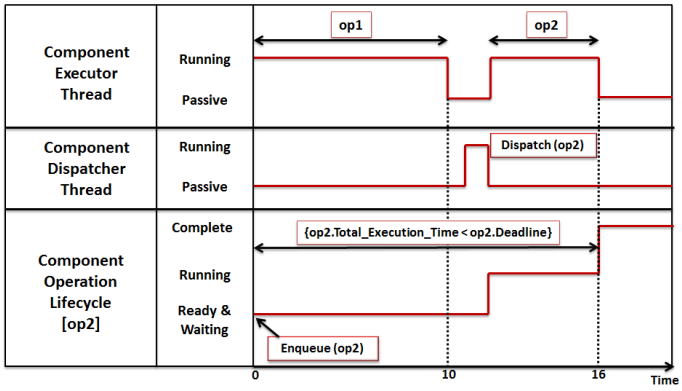

Fig. 4: Component Operation Execution

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

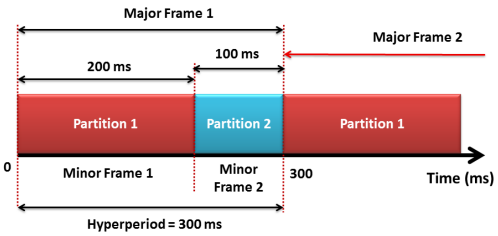

Fig. 5: Sample Temporal Partition Schedule with Hyperperiod = 300 ms

Temporal partitions are periodic fixed intervals of the CPU's time. Threads associated with a partition are scheduled only when the partition is active. This enforces a temporal isolation between threads assigned to different partitions. The repeating partition windows are called *minor frames*. The aggregate of repeating minor frames is called a *major frame*. The duration of a major frame is called the *hyperperiod*, which is typically the lowest common multiple of the partition periods. Each minor frame is characterized by a period and a duration. The period indicates how often this partition becomes active

and the duration indicates how much of the CPU time is available for scheduling the runnable threads associated with that partition. Temporal partitions with fixed periods and durations are chosen such that a valid execution schedule is realized by their coexistence. Figure 5 shows a sample temporal partition schedule.

B. 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 $(P, T, A, W, M0)$ where P is a finite set of places, T is a finite set of transitions, A is a finite set of arcs between places and transitions, W is a function assigning weights to arcs, and M0 is 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. This extended paradigm can more easily model and analyze systems with typed parameters.

IV. Problem Statement

Consider a set of mixed-criticality component-based applications that are distributed and deployed across a cluster of embedded computing nodes. Each component has a set of interfaces that it exposes to other components and to the underlying framework. Once deployed, each component functions by executing operations observed on its component message queue. Each component is associated with a single executor thread that handles these operation requests. These executor threads are scheduled in conjunction with a known set of highly critical system threads and low priority best-effort threads. Furthermore, the application threads are also subject to a temporally partitioned scheduling scheme. System assumptions include (1) knowledge on the sequence of computational steps of known duration that are executed inside each component operation, (2) knowledge on the worst-case estimated time taken on each computational step, and (3) the worst-case estimated times taken to invoke a remote function and to process a response, accounting for network-level delays. Using this knowledge about the system, the problem here is to ensure that the temporal behavior of all the application components lie within the bounds laid out by the system specifications. Ideally, this is achieved by verifying system properties like lack of deadline violations for component operations. For scenarios where the system design isn't complete, e.g. application thread priorities are unknown, the paper investigates the utility of the approach in identifying the subset of system behaviors that satisfy timing requirements and provide useful information to designers, e.g. partial thread execution orders.

V. Colored Petri Net-based Analysis Model

This section briefly describes how CPN can be used to build an extensible, scalable analysis model for component-based software applications. To edit, simulate and analyze this model, we use the CPN Tools 4 [**?**] 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 ready 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.

B. Modeling Temporal Partition Scheduling

Figure 6 shows how the temporal partitioning is modeled using CPN. Each minor frame is modeled as a record color-set consisting of a partition number, a period, a duration and an offset. Aggregate of such minor frame tokens can fully describe a partition schedule. Complete partition schedules are maintained per computing node.

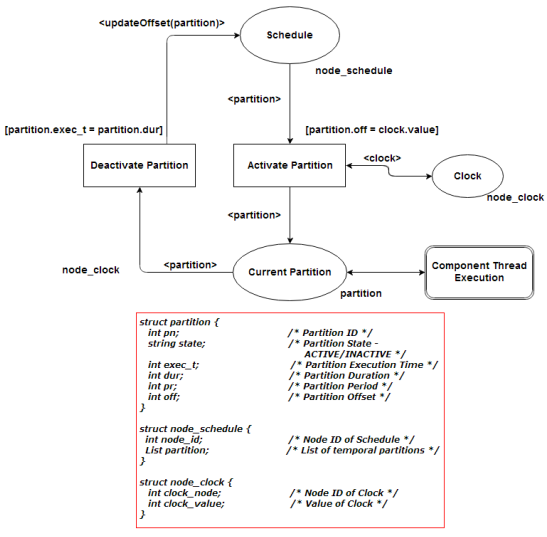

Fig. 6: CPN Model for Temporal Partition Scheduling

Assuming that the partition scheduling shown in Figure 5 is imposed on the OS scheduler in node 1, the *node_schedule*

token corresponding to this schedule is shown in Figure 7. This token populates the *Schedule* place and decides the order of partition scheduling. When the node clock reaches the offset of one of these partitions, a valid binding is realized in order to fire the *Activate Partition* which chooses the appropriate partition token and declares it as the *Current Partition*. This active partition behaves as a constraint on the set of runnable component executor threads. *Component Thread Execution* is a hierarchical transition that handles execution of component threads 1 ms at a time. At each millisecond, the *exec_t* field of the partition is updated. When this count reaches the duration of the partition, the offset of the partition is incremented by its period and the partition is made inactive. The above sequence of steps repeats for each partition.

```
1`{node_id=1, plist = [{pn=1, state=ACTIVE, exec_t=0, dur=200, pr=300, off=0},
                       {pn=2, state=INACTIVE, exec_t=0, dur=100,pr=300,off=200}]}
```

Fig. 7: CPN Token corresponding to Figure 5

C. Modeling Component Thread Behavior

Figure 8 presents a simplified version of the CPN to model the thread execution cycle. The place *Component_Threads* holds *node_threads* tokens that keep track of all the ready threads in each computing node. Each thread is a record characterized by the node ID, thread ID, partition number, priority, start time of execution and state of currently executing operation. If multiple threads belonging to the same temporal partition are ready, the highest priority thread is chosen for execution.

If the the highest priority thread is not already servicing an operation request, the highest priority operation from the *Component Message Queue* is dequeued and scheduled for execution. The scheduled thread is placed in *Currently Running Thread*. The guard on *Schedule Thread* ensures that the highest priority ready thread belonging to the *Current Partition* is always scheduled first.

When a thread token is placed in *Currently Running Thread*, the model checks to see if the thread execution has any effect on itself or on other threads. For instance: When the thread executes an operation that performs an RMI call, the effect of completion of the query is that the thread is moved to a blocked state and cannot run again till it receives a response from the server running on another thread. Such state changes are updated by the model using the *Execute Thread* transition which also progresses time by 1 ms each time it fires. Keeping track of the node clock value, the thread is preempted at each clock tick. This loop repeats as long as there are no complete deadlocks in the system.

1) Timer Operations: DREMS components are dormant by default. Once deployed, a component executor thread is not eligible to run until there is an operation added to 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-related operation is placed on the component message queue. When the component executor thread is picked by the OS scheduler, this operation is dequeued and the timer callback is executed. In CPN, timer operations are modeled as shown in Figure 9.

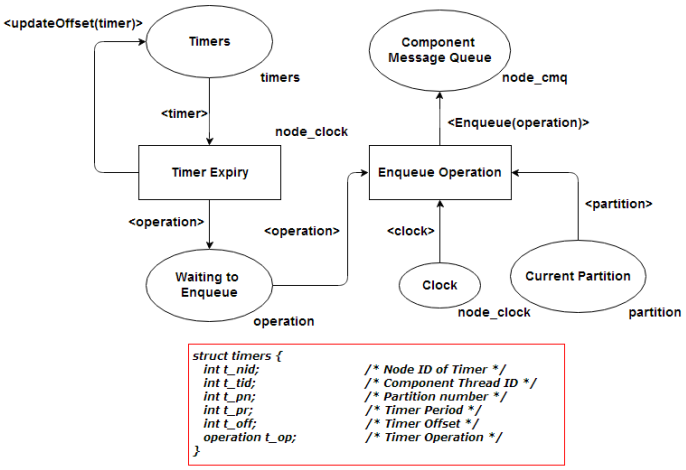

Fig. 9: Timer Operations

Every timer color-set token consists of a node ID, a thread ID, partition number, a period, an offset relative to the global node-specific clock, and an associated operation. The node ID, thread ID and partition number are necessary to correctly handle the enqueue action. All the component timers are expressed as separate tokens and initialized in the *Timers* place. Notice that the enqueue operation does not happen until the appropriate partition is active. This is because the component-specific thread responsible for enqueueing incoming operations is also affected by temporal partitioning.

D. Modeling Component Operations

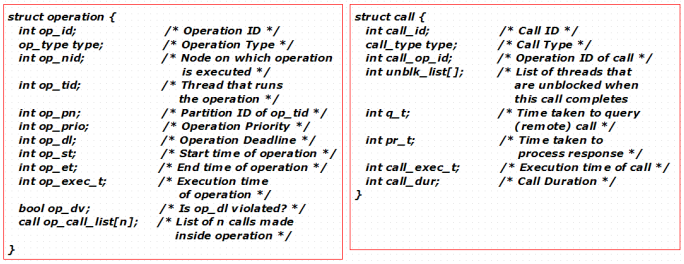

Fig. 10: Color-set for Component Operations

As shown in Figure 10, every component operation is modeled as a record color-set in CPN. This record consists of the operation ID, operation type, node ID, thread ID, partition number, priority, deadline, temporal behavior and a list of function calls that represent the business logic of the execute function. New operations are inserted into the component message queue based on priority. The time stamp *op_st* is a snapshot of the node clock value when the operation is enqueued onto the message queue. The time stamp *op_et* is a snapshot of the node clock value when the operation is completed. Accounting for the wait-time in the message queue, the total execution time of the operation is compared against its deadline *op_dl*.

Once an operation is dequeued, the execution of the operation can be treated as a transition system that runs through

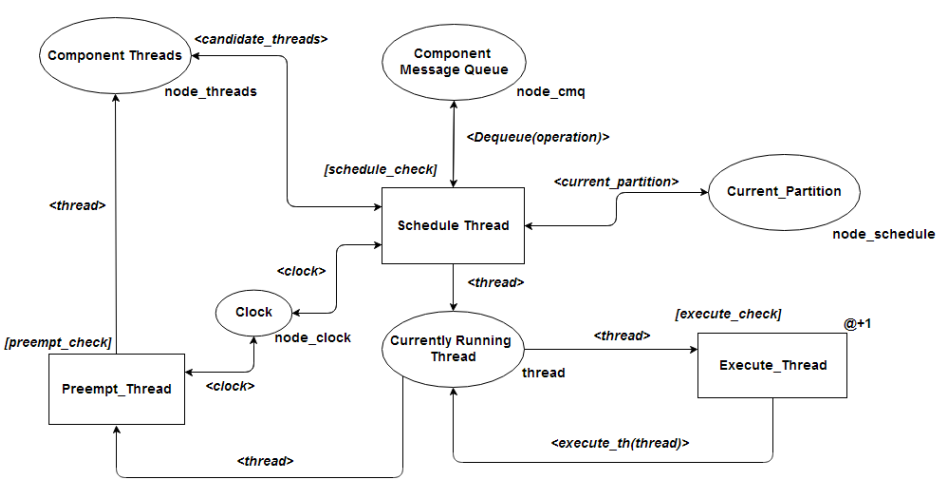

Fig. 8: Component Thread Execution Cycle

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. Consider a simple RMI application show in Figure 11. The application has two components, a client and a server. The client component is associated with a timer that triggers a sequence of interactions between the two components. When the client timer expires, a timer operation is enqueued on the client's operation queue. When scheduled, the client executor thread executes this operation, which makes an RMI call to the server component. Once the query is made, the client thread is effectively blocked till a response is received. The server thread that produces this response may not be scheduled immediately due to the constraints of temporal partition scheduling. Once the RMI operation is completed on the server, the client thread is unblocked.

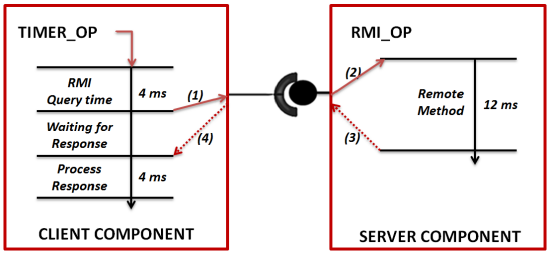

Fig. 11: RMI Application

In the above example, the duration of time for which the client is blocked, is dependent on what happens inside the remote method on the server. This remote method could either simply take up CPU time, interact with the underlying framework or interact with other components in the application. To capture such interaction patterns, the *call* color-set (Figure 10) is defined in CPN. Each call is identified by a *call_id* and a *call_type*. The field *unblk_list* is a list of threads that are unblocked when the call completes. This field is applicable to the RMI call completion on the server side which unblocks the client thread. Temporal behavior is captured using the last four fields: *q_t* is the worst-case estimate time taken for completion of a query; *pr_t* is the worst-case estimate time taken to process a response; *call_dur* is the effective duration of the call and *call_exec_t* keeps track of how much of the call is complete.

In the context of the earlier RMI example, two *operation* tokens are required to describe the operations handled by the components: a client side timer operation and a server side RMI operation. A sample client timer operation is shown in Figure 12.

```
{op_id = 1, op_type = TIMER_OP, op_nid = 1, op_tid = 1, op_pn = 1, op_prio = 50, op_dl = 80,
 op_st = 0, op_et = 0, op_exec_t = 0, op_dv = false,
 op_call_list = [{call_id = 1, call_type = RMI_c, call_op_id = 1, unblk_list = [],
                  q_t = 4, pr_t = 4,
                  call_exec_t = 0, call_dur = 8}]}
```

Fig. 12: RMI Application - Client Timer Operation

This timer operation runs on the client component (thread 1, partition 1) with a priority of 50, and a deadline of 80 ms. The business logic of this operation consists of a single RMI call that takes 4 ms to send out the query after which it blocks the executing client thread. After the client thread runs for time $t = q\_t$, the client thread is moved to a blocked state and an RMI operation is induced on the server side. The client side thread remains blocked until the server thread completes executing the remote method. Once the server thread completes execution, it sends the response of the RMI back to the client. The model takes note of how long the client has been blocked by using the time stamp at which it receives a response. The client thread runs for an additional $t = pr\_t$ to process this response before it marks completion. The token for the server RMI operation is shown in Figure 13.

This RMI operation is run on the server component (thread 2, partition 2) with a priority of 50 and a deadline of 80 ms. The deadline of this operation cannot be worse than the deadline of the client side operation that initiated the interaction. If this operation delays past 80 ms, a client side deadline violation is realized as the client thread is blocking for longer than expected.

```
{op_id = 2, op_type = RMI_OP, op_nid = 1, op_tid = 2, op_pn = 2, op_prio = 50, op_dl = 80,
 op_st = 0, op_et = 0, op_exec_t = 0, op_dv = false,
 op_call_list = [{call_id = 2, call_type = RMI_s, call_op_id = 2, unblk_list = [],
                  q_t = 0, pr_t = 0,
                  call_exec_t = 0, call_dur = 12}]}
```

Fig. 13: RMI Application - Server Operation

1) Induced Operations: To handle interactions between components, we take advantage of the developer's knowledge of the structure of the application, i.e, the component wiring. In the earlier example, we know that when the client executes an instance of a timer operation, a related RMI operation is enqueued on the server. In reality, this is handled by the underlying middleware. Since we do not model the details of this framework as of now, Figure 14 shows how the CPN model induces operations on components based on the state of the currently running thread.

Every *induced_operation* token contains a call ID and an operation. The transition *Induce* observes the activity on the currently running thread. When initiating the model, if a particular call executed by a component thread would, on completion, request the services of another component, a token is placed on the *Induce Operation* place.

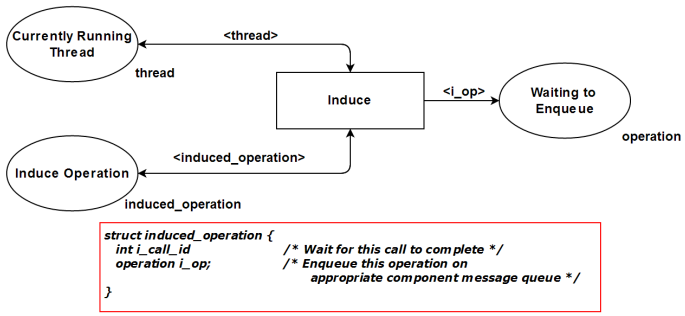

Fig. 14: Operation Induction

For the earlier RMI example, once the client thread pushes out an RMI query, an operation needs to be induced on the server queue. So an *induced_operation* token for this interaction is constructed. The model waits for the RMI call on the client side (call ID 1) to complete, at which point it places the operation *i_op* on the server message queue. This induction is represented in Figure 15.

```
{i_call_id = 1,
 i_op = {op_id = 2, op_type = RMI_OP, op_nid = 1, op_tid = 2, op_pn = 2, op_prio = 50, op_dl = 80,
  op_st = 0, op_et = 0, op_exec_t = 0, op_dv = false,
  op_call_list = [{call_id = 2, call_type = RMI_s, call_op_id = 2, unblk_list = [],
                   q_t = 0, pr_t = 0,
                   call_exec_t = 0, call_dur = 12}]}
```

Fig. 15: Operation Induction Token

In this token, *i_call_id = 1* represents the RMI call on the client. When the client thread is the currently running thread, it runs for as long as required to push out the RMI query. At this point, the *Induce* transition becomes enabled and the RMI operation *i_op* is placed in *Waiting for Enqueue*, eventually getting serviced. When initializing the model, a token is placed on *Induce Operation* for every component interaction in the system, all of which are known ahead of time when the developer designs the application.

VI. State Space Analysis

In order to describe the utility of state space-based analysis, we consider the simplified version of a trajectory planning application requiring strict temporal behavior. The component assembly for this application is shown in Figure 16. This application consists of two components: A *Sensor* component and a *Trajectory Planner* component. The Sensor component periodically publishes on a trigger topic, notifying the Trajectory Planner of the existence of new sensor data. Once the notification is received, the Trajectory Planner makes an RMI call to retrieve the data structure of sensor values. Using the updated sensor values, the Trajectory Planner calculates a new trajectory for the satellite.

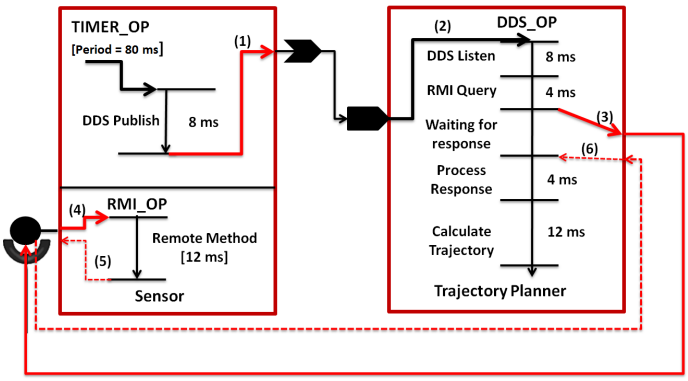

Fig. 16: Trajectory Planning Application

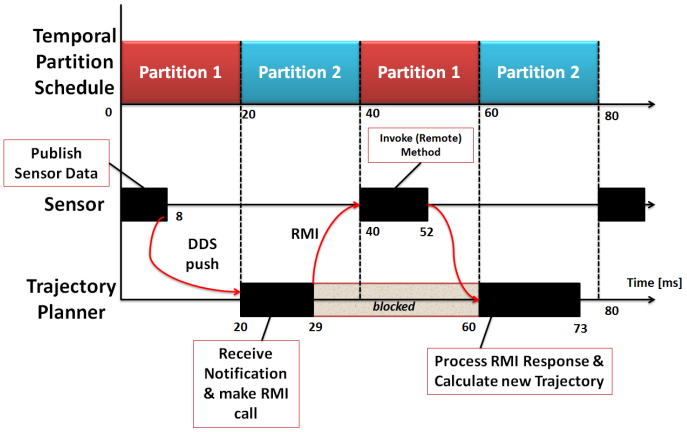

Fig. 17: Timing Diagram for Trajectory Planning

Figure 17 shows the partition schedule and temporal behavior considered. The sensor component operates on partition 1, and the trajectory planner operates on partition 2. Both partitions have a duration of 20 ms and a period of 40 ms. The sensor component is associated with a periodic timer that fires every 80 ms. When this timer expires, the sensor component publishes on a notification topic. Accounting for network latencies, the analysis assumes that this task does not take more than 8 ms. Once the notification is sent out, the sensor component becomes passive. With DDS push semantics, this notification manifests itself as a DDS operation on the trajectory planner's message queue. When partition 2

becomes active, the trajectory planner component receives the notification is has subscribed to, after which it makes an RMI call to the sensor component to obtain the updated sensor values. After the RMI call is made, this component blocks for the remainder of the partition. When the sensor component is scheduled again, it services the RMI request and sends out the RMI response, effectively unblocking the trajectory planner. Once the new sensor data is retrieved, the trajectory planner calculates a new trajectory for the satellite node.

A. Deadline Violation Detection

Using the *op_st* and *op_et* fields of every component operation, the model is capable of identifying deadline violations in component operations that are either currently in progress or waiting in the component message queue. The model essentially takes a snapshot of such cases and records the time stamps. For instance: In Figure 17, the *DDS_OP* on the Trajectory Planner Component starts at time = 20 and completes at time = 73, taking 53 ms accounting for temporal partitioning and the block time due to the remote call. If the deadline for this operation were to be set at 40 ms, the model would take notice of the violation at time = 61. This can also be observed by relying on the simulation tool as there is only one thread execution order for this scenario. Figures 18 and 19 show the observed deadline violation and the state space queries that reinforce the observation. The *SearchNodes* function enables searching parts of the state space and identifying nodes that support a predicate function. In this case, the predicate function obtains state space nodes where a deadline violation is recorded in the *Late_Operations* place. From this subset of nodes, unique deadline violations are identified. A backtrace for the observed violation can be easily obtained by using the *NodesInPath (InitialNode, DestNode)* function that presents an ordered list of state space nodes from the initial node that represents the path taken to reach the violation node.

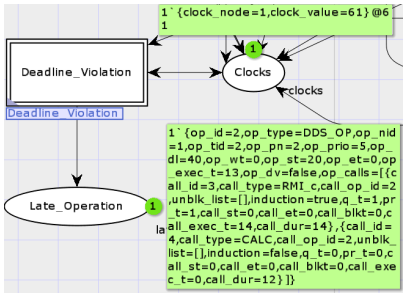

Fig. 18: Observed Deadline Violation

B. Worst-case Trigger-to-Response Time Calculation

As component operations run to completion, the analysis model keeps track of operation completion using a *Completed_Operations* place as shown in Figure 20.

For a known trigger operation and desired response operation, the worst-case trigger-to-response time can also be calculated from the generated state space. This is especially

```
val DeadlineViolation = fn : Node -> bool
val Get_Violation_List = fn : Node list -> late_opn ms list
val LateOperation_nodes =
 [99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,82,81,80,79,78,77,76,75,
  74,73,72,71,70,69,68,67,66,65,64,63,62,61,60,59,58,101,100] : Node list
val sortedLateOperation_nodes =
 [58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,
  83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101]
 : CPN'ColorSets.IntCS.cs list
val FirstDeadlineViolation =
 [{op_calls=[{call_blkt=0,call_dur=14,call_et=0,call_exec_t=14,call_id=3,
         call_op_id=2,call_st=0,call_type=RMI_c,induction=true,pr_t=1,
         q_t=1,unblk_list=[]},
        {call_blkt=0,call_dur=12,call_et=0,call_exec_t=0,call_id=4,
         call_op_id=2,call_st=0,call_type=CALC,induction=false,pr_t=0,
         q_t=0,unblk_list=[]}],op_dl=40,op_dv=true,op_et=0,
  op_exec_t=13,op_id=2,op_nid=1,op_pn=2,op_prio=5,op_st=20,op_tid=2,
  op_type=DDS_OP,op_wt=0}] : late_opn ms

fun DeadlineViolation n = (Mark.New_Page'Late_Operation 1 n <> []);

fun Get_Violation_List [] = []
  | Get_Violation_List (first_node::rest) =
   (Mark.New_Page'Late_Operation 1 first_node)::(Get_Violation_List rest);

val LateOperation_nodes = SearchNodes (
      EntireGraph,
      fn n => (DeadlineViolation n),
      NoLimit,
      fn n => n,
      [],
      op ::);

val sortedLateOperation_nodes = sort INT.lt LateOperation_nodes;
val FirstDeadlineViolation = (hd (remdupl(Get_Violation_List sortedLateOperation_nodes)));
```

Fig. 19: State Space Query for Deadline Violation Detection

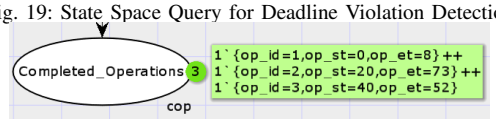

Fig. 20: Completed Operations

useful when multiple threads of same priority share a partition leading to a tree of possible thread execution orders. Once the necessary partial state space is generated, by using the operation IDs of the trigger and response, the earliest completion of the trigger operation and the latest completion of the response operation within the set period are identified. In the Trajectory Planning application, considering the *TIMER_OP* (*op_id = 1*) to be the trigger and the trajectory planning *DDS_OP* to be the response, the worst-case response time is found to be 65 ms as shown in Figure 21. Since all of the necessary information is already packed in the state space, variants of such queries can be easily constructed without changing the model.

```
val Completed_Operations =
 [[{op_et=8,op_id=1,op_st=0}],
  [{op_et=8,op_id=1,op_st=0},{op_et=52,op_id=3,op_st=40}],
  [{op_et=8,op_id=1,op_st=0},{op_et=73,op_id=2,op_st=20},
   {op_et=52,op_id=3,op_st=40}]] : cop ms list
val Trigger_Operation = 1 : int
val Response_Operation = 2 : int
val trigger_earliest_et = 8 : CPN'ColorSets.IntCS.cs
val response_latest_et = 73 : CPN'ColorSets.IntCS.cs
val Trigger_to_Response_Time = 65 : CPN'ColorSets.IntCS.cs

val Completed_Operations = (remdupl(Get_coplist (sort INT.lt
      (SearchNodes ( EntireGraph, fn n => (isCompleted n),
                     NoLimit, fn n => n, [], op ::)))));
val Trigger_Operation = 1; val Response_Operation = 2;
val trigger_earliest_et = hd(sort INT.lt (remdupl
                          (get_etlist Trigger_Operation Completed_Operations)));
val response_latest_et = hd(rev(sort INT.lt (remdupl
                          (get_etlist Response_Operation Completed_Operations))));
val Trigger_to_Response_Time = response_latest_et - trigger_earliest_et;
```

Fig. 21: Worst-Case Trigger-to-Response Time Calculation

C. Partial Thread Execution Order Generation

Since the start and end time stamps of any completed operation can be obtained from the state space, this feature can be exploited to obtain a partial thread execution order

and therefore thread priorities for applications in the design process. Consider the sample application shown in Figure 22.

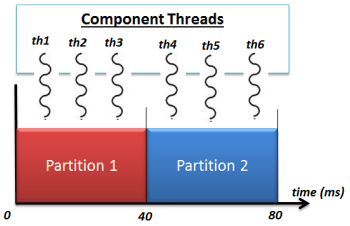

Fig. 22: Sample Application in Design Process

This application consists of 6 component executor threads that service component operation requests. 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 associated with a timer that triggers an operation once every major frame, taking up to 8 ms to complete. The design requires that operation 3 (handled by thread 3) must complete before 20 ms and operation 5 (handled by thread 5) must complete before 60 ms from start of schedule. Figure 23 shows queries that use the generated state space to obtain a thread execution order that satisfies such timing constraints. All threads are assigned the same priority and scheduling is therefore Round-Robin. The generated state space (size = 5000 nodes) records all possible thread execution orders for one hyperperiod of the schedule. From this partial state space, the state space node that first satisfies the timing requirement is identified. By generating a backtrace from the initial node and identifying the subset of nodes where the system clock ticks, the thread execution order is determined. It is clear that as the timing requirements become stricter, the number of possible thread execution orders decrease. For multiple timing requirements, if the set of thread execution orders contradict each other, the order that satisfies the higher priority timing requirement is chosen.

```
val desired_SS_node = 3962 : Node
val Partial_Order =
  [{th_st=0,th_tid=3},{th_st=4,th_tid=1},{th_st=8,th_tid=2},
   {th_st=12,th_tid=3},{th_st=16,th_tid=1},{th_st=20,th_tid=2},
   {th_st=40,th_tid=5},{th_st=44,th_tid=6},{th_st=48,th_tid=4},
   {th_st=52,th_tid=5},{th_st=56,th_tid=6},{th_st=60,th_tid=4}]
  : {th_st:INT, th_tid:INT} list

(* Predicate is true if op_id1 completes before 20 ms and op_id2 completes before 60 ms*)
fun Predicate1 op_id1 op_id2 (first_cop::rest) =
      if ((get_et op_id1 first_cop <> 0) andalso (get_et op_id1 first_cop < 20) andalso
          (get_et op_id2 first_cop <> 0) andalso (get_et op_id2 first_cop < 60) andalso
            (length first_cop = 6)) then
             first_cop else
           Predicate1 op_id1 op_id2 rest;

val desired_SS_node = Get_copnode (Predicate1 3 5 Completed_Operations)
                                                  Completed_Operation_nodes;
val Partial_Order = remdupl(get_th_execution_order
                          (get_clock_tick_nodes (NodesInPath (1, desired_SS_node))));
```

Fig. 23: Partial Order for Thread Execution

D. Scalability Testing

The size of the generated state space is dependent on the amount of concurrency in the behavior. If all the executing threads had unique priorities, the thread execution order is a constant as the scheduling is priority-based. However, for larger systems with multiple applications and with threads of same priority sharing processor time in the same partition, several thread execution orders are possible depending on the arrival rates of operation requests and state of executing threads.

Consider a set of mixed-criticality applications deployed on a 3-node satellite cluster. Each node has a unique temporal partition schedule with the longest schedule having a hyper-period of 1 second, spanning 5 partitions. Hundred threads within the priority range [30-90] are distributed across the nodes. Most of the application component threads are triggered by timers of varying periodicity. In order to maximize the concurrency and possibilities of thread execution orders, most of the application threads are assigned the same priority and are also completely independent of each other. Dependence between components forces certain thread execution behaviors and so this is avoided. The Analysis model has 14 transitions and 11 places of which 3 places are observer places. Structural reductions to the model further reduce the state space size. Table I summarizes the results of this effort. One of the observed consequences for such large systems is that the CPN Tools user interface suffers a rendering lag when it has to keep track of and display large token sets. This can be avoided by hiding as much detail as possible using hierarchy.

TABLE I: Scalability Testing

Satellites	Threads	Hyperperiods	State Space Nodes
1	50	10	125,000
3	100	10	480,000

VII. Analysis Model Generation

For large applications, with several timers and component interaction patterns, hand-writing the CPN token specification will prove to be cumbersome and error prone. This can be avoided by integrating the temporal behavior specification for component-based applications onto the modeling framework used to generate the deployment plans and infrastructure glue code, effectively closing the loop shown in Figure 1. Since the structure of the places, transitions, color-sets, variables and function declarations do not change, only the application-specific token structure needs to be derived from the design model of the application. Figure 24 shows a part of the ANTLR-based [**?**] specification grammar, a part of the input text file and the generated CPN tokens.

VIII. Discussion

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 for a designer is dependent heavily on the initial design itself. With increasing number of timer operations and equal priority threads sharing a CPU, the number of possible thread execution orders and behavioral scenarios will exponentially grow, leading to unmanageable state space sizes. The results shown in Table I do not represent an upper bound on the state space size but one corresponding to an average-case scenario. For safety-critical systems, deterministic system behavior requires deterministic

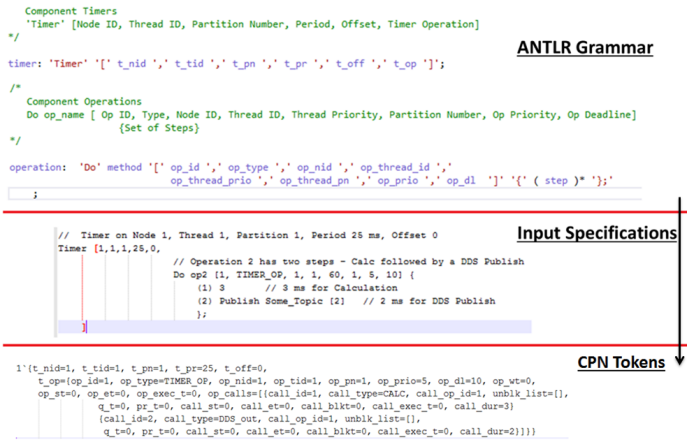

Fig. 24: Analysis Model Generation

system designs. Since worst-case estimates are considered, it is unlikely that the actual system behavior would even reach certain behavioral regions. Therefore, the analysis results obtained from this approach effectively behave as guidelines for a more refined, and predictable application design as opposed to hard constraints on possible design.

IX. Future Work

In order to generalize this analysis model and provide flexibility, one possible extension to this approach is to cater to other commonly used scheduling schemes such as EDF, RMS etc. for component operation scheduling; and novel interaction patterns (e.g. reliable broadcast). Since the analysis model is a composition of sub-nets that are not tightly coupled, this extension would provide reusable, self-contained functional modules.

The current analysis approach inherits the benefits and the drawbacks of using pessimistic estimates for execution times. Another possible extension to this approach would be to provide a stochastic schedulability analysis allowing for a trade-off between reliability and cost of resources required by the system. However, describing execution times for software using probability distribution functions is not trivial and would require significant effort.

X. Conclusions

Mobile, distributed real-time systems operating in dynamic environments, and running mission-critical applications face 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. This analysis model includes a compact, scalable representation of high-level design, accounting for the dynamics of real-time thread execution while exploiting knowledge of component execution code. Exhaustive state space search enables verification and validation of useful and necessary system properties, reducing development costs and increasing reliability for such time-critical systems. The utility of this tool has been illustrated with several examples.

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.