Colored Petri net-based Modeling and Schedulability Analysis of Component-based Applications

Pranav Srinivas Kumar, Gabor Karsai, Abhishek Dubey
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235
{pkumar, gabor, dabhishe}@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 schedulability analysis of component-based software that maps a high-level design model onto abstractions enabling the verification of useful system properties, e.g. lack of deadline violations. This approach also addresses the need to model the temporal behavior of the applications under some level of abstraction and integrate it into the design of the underlying components. The developer's knowledge on the component execution code will further enable validation of scheduling constraints that are inherent in such real-time systems.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software/Program Verification—*Reliability*

General Terms

Verification

Keywords

Colored Petri Nets, Modeling, Schedulability, Verification

1. 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. Component models describe what these component blocks are, 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 profile 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 significant access to such remote systems for run-time debugging and evaluation. These types of systems, therefore demand the need for advanced design-time modeling and analysis methods to detect possible anomalies in system behavior, such as missed application deadlines, before deployment.

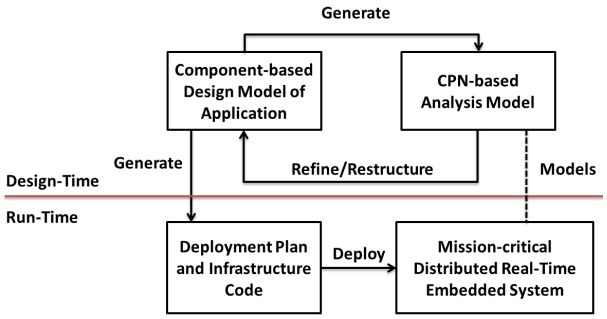


Figure 1: Verification-driven Workflow

Figure 1 shows a Verification-driven workflow for component-based software development. Application developers use domain-specific modeling tools to model the component assembly, interaction patterns, component execution code, sequence of operations, and associated temporal properties such as estimated execution times, deadline etc. Using such application-specific parameters in the design model, a Col-

ored Petri net-based (CPN) [1] analysis model is generated. The system behavior is analyzed to check properties like lack of deadlocks, deadline violations etc. The results of this analysis help improve the structure of the application, enabling safe deployment of dependable components that are known to operate within system specifications. In this paper, we present the CPN-based approach that enables schedulability analysis for component-based applications.

The remainder of this paper is organized as follows. Section 2 presents existing research relating to this paper and explains how this approach is different; Sections 3 and 4 provide a brief description of the component model, the operating system scheduler and the associated infrastructure considered to illustrate the approach; Section 5 elaborates on how this architecture is abstracted and modeled using Colored Petri-nets. This section also describes the analysis of the generated state space to answer schedulability; Sections 6 and 7 present future extensions to the proposed approach and concluding remarks respectively.

2. 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. The design models are transformed into relevant analysis models to study probable behaviors of the system and identify anomalies. Using the analysis results, the design is refined and safe, predictable system behavior is achieved.

Petri nets and their extensions have proved to be a powerful formalism for modeling and analyzing real-time systems. The authors in [2] analyze the behavior of AADL models using Petri net extensions, checking for the validity of system properties. For the verification of real-time properties such as missed deadlines, the authors proposes using Timed Petri nets (TPN), focusing on periodic threads in systems that are not preemptive. The analysis is illustrated with a sample application consisting of communicating periodic threads running on a single processor.

For accurate analysis of component-based software operating on real-time systems, it is necessary to obtain significant information about the component assembly, the interaction patterns and real-time data about the temporal behavior of components. [3] addresses this strategy integrating application configuration parameters, thread priorities, scheduling policies etc. onto the component descriptors. This approach, however does not consider knowledge of the component's execution code.

Several analysis approaches present tool-aided methodologies that exploit the capabilities of existing analysis and verification techniques. In the verification of timing properties of composed systems, [4] uses the OMG UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) modeling standard with model transformations to convert design models into MAST output models for concrete schedulability analysis. Similarly, [5] describes an approach for formal schedulability analysis of AADL models. The AADL model is translated into the real-time process algebra ACSR. The problem of schedulability is reduced to deadlock detection, which is performed by searching the model state space using VERSA. The analysis also presents failing scenarios in case of violations.

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 [6] 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.

For classes of component-based systems whose component assembly and application structure change dynamically over time, design-time verification is not sufficient. The work presented in [7] focuses on reverification for such dynamic systems. The paper describes the INcremental VErification STratery (INVEST) framework that augments with compositional verification and identifies the minimal set of components that require reverification after dynamic changes to the underlying system.

In our CPN-based analysis approach, developers build component based applications using modeling tools to specify the component assembly, the interaction patters, the sequence of operations, and the associated temporal properties such as estimated execution times. The execution code of the components provided by the developer are abstracted into a suitable form and used as parameters in the CPN-based analysis model. The behavior of the system is analyzed when operating on a hierarchical scheduling scheme. The result of this analysis helps identify anomalies and enables restructuring of the application for performance within the system specifications

3. THE COMPONENT MODEL

3.1 DREMS Platform

Our team has designed and prototyped a full information architecture called **D**istributed **RE**al-time **M**anaged **S**ystem (DREMS) [8, 9] that addresses requirements for rapid component based application development, with multiple interaction semantics, concurrency, resource management, application security, isolation; and managed deployment and configuration. This architecture consists of a design-time tool suite for modeling, analysis, synthesis, integration, debugging, testing, and maintenance of application software built from reusable components. The architecture also provides a run-time software platform for deploying, managing, and executing application software on a network of mobile computing nodes. The run-time software platform consists of an operating system kernel, system services and middleware libraries. In prior work, we have described the general architecture [8, 9], the design-time modeling capability [10], and the component model used to build applications [11].

The DREMS platform is characterized by a hierarchical scheduling scheme. The higher, component-level scheduler schedules component operations for execution, enabling interac-

tions between application components. The lower, operating system scheduler enforces a temporal partitioning scheme and schedules the component threads. Components are grouped into unique, identifiable processes called actors. These actors are assigned to temporal partitions and run only when the associated temporal partition is active. This partitioned scheduling provides a guaranteed slice of the CPU to all the actors and enables isolation between multiple applications running on the same computing node.

3.2 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 business logic that implements operations on state variables. Ports enable interactions between communicating components. A component-level message queue, with associated infrastructure code, schedules operations on the individual components. Timers can be configured to trigger operations and management code can be used to detect faults, deadline violations etc. Figure 2 shows a basic DREMS component.

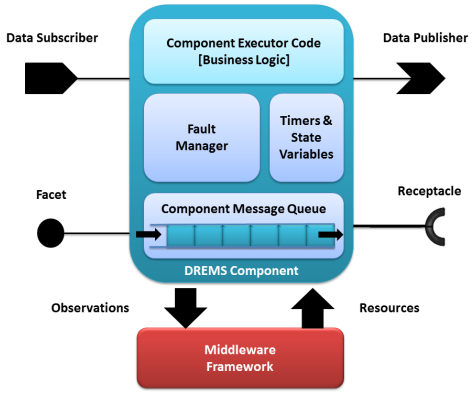


Figure 2: DREMS Component

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.

Additionally, this component model also supports sporadic and periodic timers that can be used to initiate component operations. More details on this component model can be found in [11].

3.3 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 operations 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).

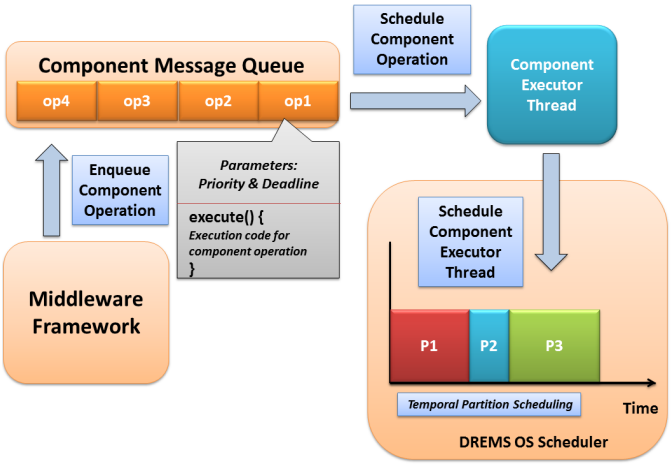


Figure 3: Scheduling Component Operations

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. These operations are sorted and scheduled based on one of three scheduling schemes: Earliest Deadline First (EDF), First In First Out (FIFO) and Priority FIFO.

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 the 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 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 oper-

ation, *op1*. At this point, consider that a new component operation *op2* is enqueued on the message queue and marked as ready. The executor thread does not become passive until time 10, at which point the component dispatcher thread dequeues the next ready operation for execution. Assuming the component executor thread is scheduled immediately by the underlying processor, this thread runs the ready operation by invoking its *execute* function. This operation is run to completion at time 16. The total time taken for execution of this operation is measured from when the operation was enqueued, i.e. time 0. If the time taken for the operation exceeds its deadline, a fault manager is immediately notified. The duration of the component operation is further delayed by temporal partitioning enforced by the OS scheduler. This adds to the need for schedulability analysis, especially in case of safety and mission-critical applications.

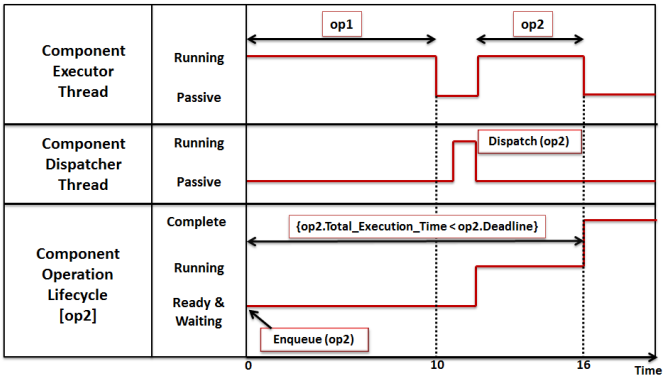


Figure 4: Component Operation Execution

4. TEMPORAL PARTITION SCHEDULER

The DREMS OS scheduler was implemented by modifying the behavior of the standard Linux scheduler, introducing an ARINC-653 [12] style temporal and spatial partitioning scheme. The scheduler further groups threads into different criticality levels: Critical, Application and Best-Effort. Critical threads handle system and mission management tasks; Application-level threads include component executor threads that perform mission-specific non-critical tasks; Best-Effort threads handle low priority tasks that are scheduled only when there are no other runnable threads from the other two categories.

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. This schedule is made up of two minor frames. Assuming the schedule is enforced at time t = 0, partition 1, characterized by minor frame 1, is made active by the partition scheduler. This partition stays active for the duration of the minor frame, 200 ms. At this point, partition 1 is deactivated and partition 2, characterized by minor frame 2, is activated. Partition 2 stays active for 100 ms, after which the major frame ends and the schedule is repeated.

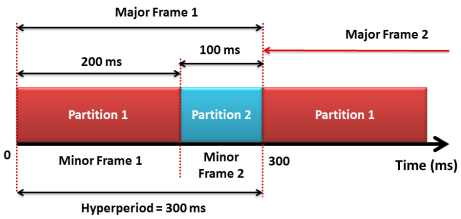


Figure 5: Sample Temporal Partition Schedule

5. COLORED PETRI NET-BASED ANALYSIS MODEL

Petri nets [13] 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. Since petri net tokens are not distinguishable, the obtained model is often too complex for large systems. To enable compact representations for the modeled system, extensions to the basic petri net model, called high-level petri nets, are preferred.

With Colored Petri nets (CPN) [1], 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.

This section briefly describes how CPN can be used to build an extensible, scalable analysis model for component-based software applications. In the context of the component model and the OS scheduler described in Sections 3 and 4, this involves capturing (1) the nature of temporal partitioning, (2) properties of the component executor threads, (3) priority-based scheduling of the component executor threads, and finally, (4) using knowledge of the component's business logic code to model the scheduling of component operations. To edit, simulate and analyze this model, we use the CPN Tools 4 [14] tool suite.

5.1 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 to schedule thread execution from a queue of ready threads. The highest priority thread is always chosen first for scheduling. 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, a Round-Robin (RR) scheduling algorithm 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. Since thread quanta are typically measured in clock ticks, we have chosen the temporal resolution to be 1 clock tick of the system timer. Therefore, in all of the modeling aspects that follow, temporal behavior of component operations is measured in *ticks*.

5.2 Modeling Temporal Partitioning

As described in Section 4, a temporal partition schedule is represented by a major frame comprised of a sequence of minor frames. Each minor frame specifies the execution time of an individual partition for a fixed duration. At the end of each major frame, the partition schedule repeats.

Figure 6 shows how temporal partitioning can be modeled using CPN. Each minor frame is treated as a record color-set consisting of an identity, a period, a duration and an offset. The identity is used to indicate the ID of the associated partition. The period, duration and offset of the minor frame are each measured in ticks. Aggregate of such minor frame tokens can fully describe a partition schedule.

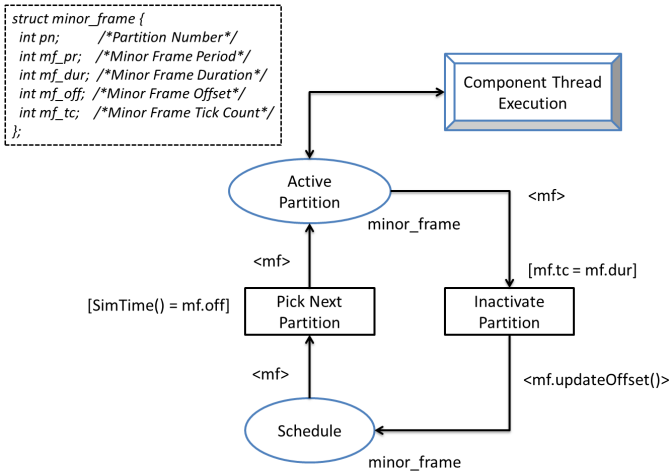


Figure 6: CPN model for Partition Scheduling

Assuming the duration of a clock tick to be 4 ms, the sample schedule in Figure 5 requires the following two minor frame tokens:

MinorFrame 1 = [pn = 1, mf_pr = 75, mf_dur = 50, mf_off = 0, mf_tc = 0]
MinorFrame 2 = [pn = 2, mf_pr = 75, mf_dur = 25, mf_off = 50, mf_tc = 0]

Minor frame 1 has a duration of 200 ms and period of 300 ms which is 50 and 75 clock ticks respectively. Similarly, minor frame 2 has a duration of 100 ms and a period of 300 ms which is 25 and 75 clock ticks respectively. These tokens populate the *Schedule* place and decide the order of partition scheduling. When the global clock reaches the offset of one of the minor frames, a valid binding is realized in order to fire *Pick Next Partition* which chooses the appropriate minor frame token and declares it as the *Active 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 tick at a time. At each tick, the *tick_count* field of the minor frame is updated. When this count reaches the duration of the minor frame, the offset of the frame is incremented by its period and the frame is made inactive. The above sequence of steps repeats for each minor frame.

5.3 Modeling Component Thread Behavior

Figure 7 presents a simplified version of the CPN model to model the thread execution cycle. The place *Component Threads* holds a list of *thread* tokens to keep track of all the ready threads in the node. Each thread is a record characterized by the thread's ID, partition ID, priority, state and operation queue. The state of every thread can either be ready, running or blocked. If the operation queue is non-empty, the thread is in the ready state. If multiple threads belonging to the same temporal partition are ready, the highest priority thread is chosen for execution. Depending on the currently active partition and the priority of ready threads belonging to this partition, one of the threads from the *thread_list* enables the *Pick Next Thread* transition. This thread moves to the place - *Currently Running Thread*. The guard *Exec_Guard* ensures that the thread chosen belongs to the active partition. This captures the temporal partitioning enforced by the OS scheduler.

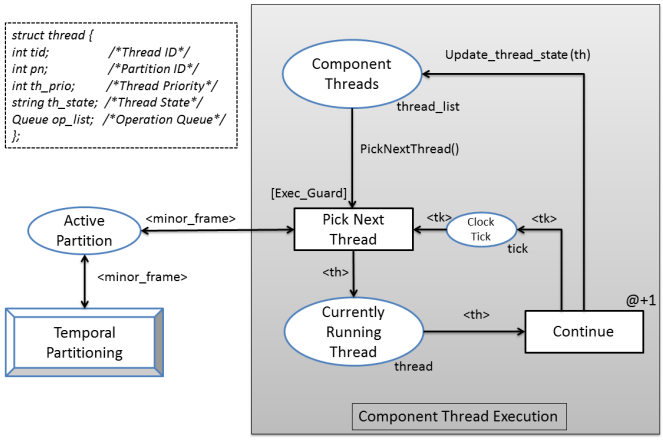


Figure 7: Component Thread Execution Cycle

When a thread token is placed in *Currently Running Thread*, the model checks to see if this 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 this 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 before the transition *Continue* can fire. *Continue* is a timed transition that

progresses the global time by 1 clock tick. When this transition fires, the thread state is updated and the thread is appended back into the list of ready threads and the cycle is repeated.

5.3.1 Timer Operations

DREMS components are dormant by default. Once deployed, a component executor thread does not run until there is an operation added to the message queue.

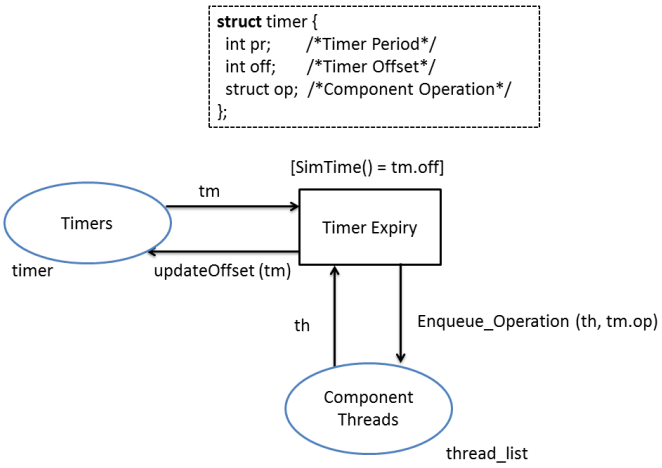


Figure 8: Timer Operations

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 is placed on the queue. When the component's 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 8.

Every *timer* color-set token consists of a period, an offset relative to the global simulation time and an associated operation. All of the component timers are expressed as separate tokens and initialized in the *Timers* place. When any of the component timers expire, the component thread is extracted from the set of threads, and the timer operation is enqueued on the component's *op_list*. The *Timer Expiry* transition can fire multiple times when more than one timer expires at the same time. The offset of the timer token is updated before it is placed back into *Timers*.

5.4 Modeling Component Operations

```
struct operation {
int op_tid;      /*Thread that runs the operation*/
int op_pn;       /*Partition ID of op_tid*/
int op_id;       /*Operation ID*/
int op_prio;     /*Operation Priority*/
string op_type;  /*Type of Operation*/
int op_dl;       /*Operation Deadline*/
int op_st;       /*Operation start time*/
int op_et;       /*Operation end time*/
int op_wt;       /*Operation wait time*/
call op_call_list[n];  /*List of n calls made in op*/
};
```

```
struct call {
int call_id;     /*Call ID*/
string call_type; /*Type of call*/
int blk_list[];  /*List of Threads that are
                   blocked when call completes*/
int unblk_list[]; /*List of threads that are
                    unblocked when call
                    completes*/
int call_qt;     /*Time taken to query (remote)
                   call*/
int call_prt;    /*Time taken to process
                   response*/
int call_dur;    /*Effective duration of call*/
int call_tc;     /*Tick_count for call progress*/
```

Figure 9: Color-set for Component Operations

As shown in Figure 9, every component operation is modeled as a record color-set in CPN. This record consists of the ID of the thread that executes this operation, the partition ID of the executing thread, the operation's ID, 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 global clock when the operation is dequeued for execution. The time stamp *op_et* is a snapshot of the global clock when the operation is completed. Accounting for the wait-time *op_wt* 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 a sequence of function calls dictating its behavior. Any of these underlying function calls can have an state-changing effect on the thread executing this operation. Consider the simple RMI application shown in Figure 10. 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 partitioning. Once the RMI operation is completed on the server, the client thread is unblocked.

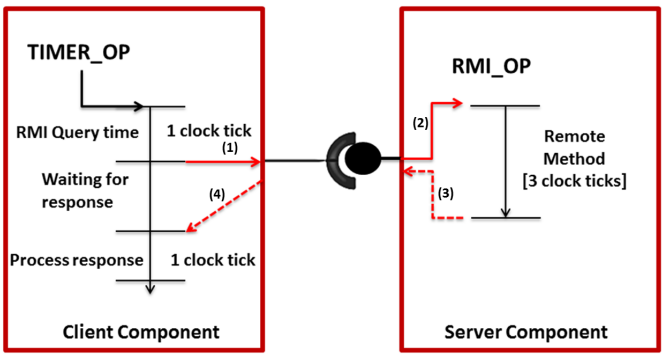


Figure 10: RMI Application

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 9) is defined in CPN. Each call is identified by a *call_id* and *call_type*. The field *blk_list* is a list of threads that are blocked when this call completes. In case of an RMI call, blk_list contains the thread ID of executing thread. Similarly, 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: *call_qt* is the time taken for completion of a query; *call_prt* is the time taken to process a response; *call_dur* is the effective duration of the call, managed by the model and measured in ticks.

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 expressed as follows:

```
{op_tid = 1, op_pn = 1, op_id = 1, op_prio = 50, op_type = TIMER_OP, op_dl = 20,
 op_st = 0, op_et = 0, op_wt = 0,
  op_call_list = [{call_id = 1, call_type = RMI_c, blk_list = [1], unblk_list = [],
                   call_qt = 1, call_prt = 1,
                   call_dur = 2, call_tc = 0}]}
```

This timer operation runs on the client component (thread 1, partition 1) with a priority of 50, and a deadline of 20 clock ticks. The business logic of the operation consists of a single RMI call that takes 1 tick to send out the query after which it blocks the executing client thread. After the client thread runs for time t = call_qt, 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 this RMI operation. 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 blocking for using the time stamp at which it receives the response. The client thread runs for an additional time t = call_prt to process this response before it marks completion. The token for the server RMI operation would look like this:

```
{op_tid = 2, op_pn = 2, op_id = 2, op_prio = 50, op_type = RMI_OP, op_dl = 20,
 op_st = 0, op_et = 0, op_wt = 0,
  op_call_list = [{call_id = 2, call_type = RMI_s, blk_list = [], unblk_list = [1],
                   call_qt = 0, call_prt = 0,
                   call_dur = 5, call_tc = 0}]}
```

This RMI operation is run on the server component (thread 2, partition 2) with a priority of 50 and a deadline of 20 ticks. The deadline of this operation cannot be worse than the deadline of the client side operation that started the interaction. If this operation delays past 20 ticks, a client side deadline violation is realized as the client thread is blocking for longer than expected.

5.4.1 Operation Induction

To handle interactions between components, we take advantage of the developer's knowledge on 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 enqueue is handled by the underlying framework. Since we do not model this framework as of now, Figure 11 shows how the CPN model induces operations on components based on the state of the currently running thread.

Every *i_op* token contains a call ID and an operation. The transition *Induce* observes the activity on the currently running thread. When initializing 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.

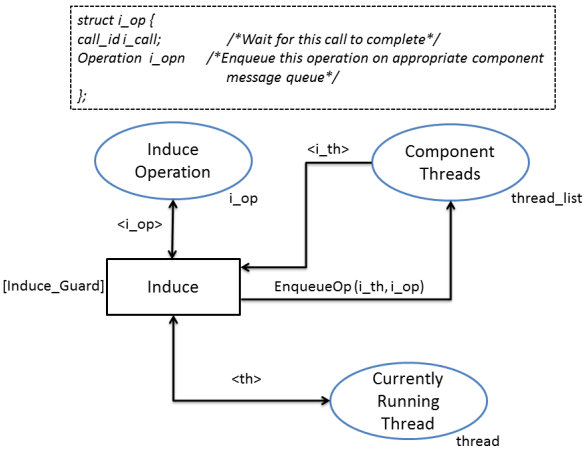


Figure 11: Operation Induction

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 *i_op* 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_opn* on the server message queue. This induction is represented by the following token:

```
{i_call = 1,
 i_opn = {op_tid = 2, op_pn = 2, op_id = 2, op_prio = 50, op_type = RMI_OP, op_dl = 20,
  op_st = 0, op_et = 0, op_wt = 0,
   op_call_list = [{call_id = 2, call_type = RMI_s, blk_list = [], unblk_list = [1],
                    call_qt = 0, call_prt = 0,
                    call_dur = 5, call_tc = 0}]}}
```

In this token, i_call = 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 in time, the *Induce* transition becomes enabled and the RMI operation i_opn is enqueued on the server thread. Similarly, a token is placed on *Induce Operation* for every component interaction, all of which are known ahead of time when the developer designs the application model.

5.5 Simulation and State Space Analysis

In order to describe the utility of simulation- and state space-based analysis, we consider the simplified version of a trajectory planning application for a satellite. This is a mission-critical application requiring strict temporal behavior. The component assembly for this application is shown in Figure 12. 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.

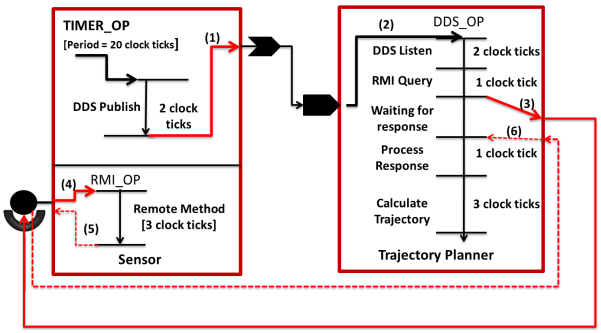


Figure 12: Trajectory Planning Application

Figure 13 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 5 clock ticks and a period of 10 clock ticks. The sensor component is associated with a periodic timer that fires every 20 clock ticks. 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 2 clock ticks. Once the notification is sent out, the sensor component becomes passive. With DDS push semantics, this notification manifests as a DDS operation on the trajectory planner's operation queue. When partition 2 becomes active, the trajectory planner component receives the notification it 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. For the sake of analysis, we assume the temporal behavior shown in Figure 12.

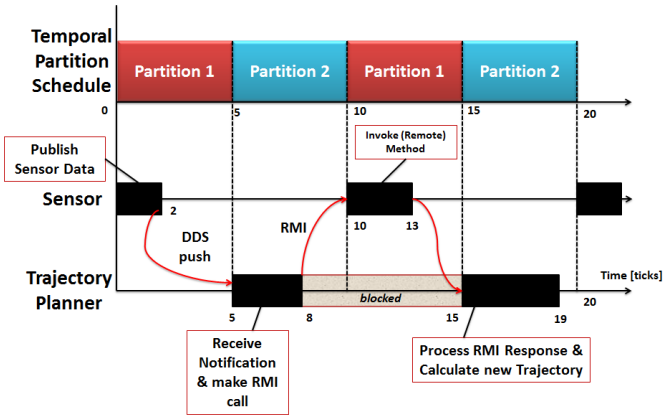
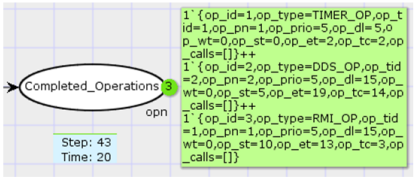


Figure 13: Timing Diagram for Trajectory Planning

To initialize the CPN model, the partition schedule, component assembly and temporal behavior are abstracted into appropriate tokens: minor_frame tokens in the *Schedule* place, thread tokens in the *Component Threads* place, timer tokens in the *Timers* place, and finally i_op tokens in *Induce Operations* place to handle the component interactions. The operation induced on the trajectory planner consists of three *calls*: a DDS listen call to receive notification, a consequent RMI call to the sensor, and a *CALC* call taking up CPU for 3 clock ticks to indicate computation of new trajectory.

5.5.1 *Simulation*

Using the simulation tool in CPN Tools, the system is simulated one step at a time. The temporal behavior of the two component threads can be analyzed to see if there are any deadline violations or deadlocks. The simulation can also be fast forwarded by a finite number of steps, or time units. This enables hiding transition firings and jumping to a future point in time to check system properties. For instance: Figure 14 shows the token marking on the *Completed_Operations* place, 20 clock ticks from initialization.



COMPONENT OPERATION	START TIME [Time-stamp]	END TIME [Time-stamp]	DEADLINE [Ticks]
TIMER_OP [on Sensor]	0	2	5
DDS_OP [on Trajectory Planner]	5	19	15
RMI_OP [on Sensor]	10	13	15

Figure 14: Trajectory Planning: No Deadline Violations

The trajectory planner thread avoids a deadline violation by completing its task in 14 clock ticks, as illustrated in Figure 13. If the deadline for this operation was set to 10 clock ticks, the simulation of the model stops when the global clock hits 15 clock ticks, as shown in Figure 15, as it has detected a deadline violation - The trajectory planning operation, which started at time stamp 5, does not complete when model time reaches time stamp 15.

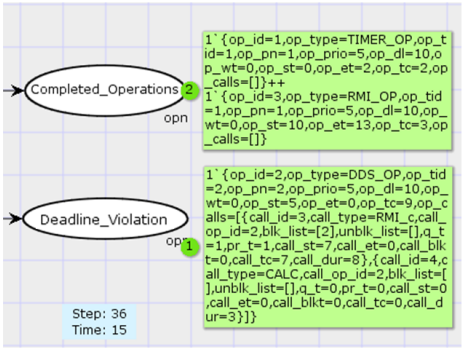


Figure 15: Trajectory Planning: Deadline Violation

5.5.2 *State Space Analysis*

Simulation-based analysis of system behavior is dependent on the simulator tool choosing a unique path from within a

tree of possible paths the system takes, and hoping that the chosen path leads to the detection of property violations. Since the number of possible trajectories in complex, concurrent, distributed systems grows exponentially with size, the reliability of simulation-based analysis reduces, therefore necessitating a search through the system state space (SS) to detect property violations in any of the possible trajectories and to guarantee schedulability.

Using the in-built state space tool, a complete or partial state space can be generated for the modeled system. Partial state spaces are generated by specifying predicates or limiting parameters like number of SS nodes, steps, seconds etc. These limiting parameters help keep the analysis model scalable for larger systems by generating a smaller, truncated state space. From the generated state space, a standard report is derived in text form. This report provides the information on: (1) Size of the state space, with number of nodes, arc, seconds etc., (2) Boundedness Properties for all the places in the model, (3) Liveness Properties for all the transitions in the model, (4) Fairness Properties for transition firings, and (5) Home Properties for home markings. Boundedness properties can be used to check the upper limit on the size of place markings. Ideally, the developer would like to see the upper limit for the *Deadline_Violation* place to be zero tokens. Liveness properties can also be assessed to check for deadlocks - If the *Pick_Next_Thread* transition is marked as a dead transition, this could either mean that all component threads are ready and passive or that all component threads are blocked, leading to a deadlock.

Besides the generated report, state space queries can be written as auxiliary text in the model and evaluated. For ex: *TIsDead ([TI.Thread_Execution'Pick_Next_Thread 1], 1);* is a simple boolean query that checks to see if there exists an occurrence sequence from Marking 1 where the transition instance Pick_Next_Thread belonging to the Thread_Execution page becomes dead. Alternately, all of the dead transitions from the state space can be aggregated and printed as a list using the simple *ListDeadTIs();* query.

Non-standard queries based on the in-built search function are useful to detect deadline violations in the generated state space. The *SearchNodes* function performs calculations on each SS node and the results are combined to form the final output. This query takes a few different arguments: (1) Search Area, representing how much of the state space needs to be checked, (2) a predicate function to avoid unnecessary paths, (3) an integer limit for the number of times the predicate evaluates to true before terminating [In case of deadline violation, this can be just 1], (4) an evaluation function, mapping state space nodes to values, and finally (5) some combination function to combine the results of all the positive results.

```
val it = [1] : Node list

SearchNodes (EntireGraph,
             fn n => size(Mark.Analysis_Model'Deadline_Violation 1 n) > 0,
             1,
             fn n => n,
             [],
             op ::)
```

Figure 16: SearchNodes Function

A sample query can be seen in Figure 16. This query searches the entire state space graph, checking to see if the marking on the place *Deadline_Violation* in page Analysis_Model is greater than zero. The search will terminate after the first positive result. The evaluation function does not compute anything from these nodes and so the evaluation is a map function *fn n => n*. The combination function *op::* appends positive results of the search together and provides as a final result, a list of state space nodes in which the predicate is true. Predecessors of the output state space nodes can be evaluated to realize how a deadline violation was achieved from the initial root node of the state space.

Since complex data structures can be used to represent the component behavior, using CPN leads to a compact analysis model. The CPN model for the above example consisted of 12 places and 11 transitions, capable of modeling various component interactions such as RMI, AMI, DDS and timer-based operations. For larger systems with hundreds of components, timers and interactions, the structure of the analysis model remains the same, with additional tokens representing the different component interactions and behaviors. Though the state space generation for the Trajectory planning application took less than 3 seconds, for larger, distributed systems the number of concurrent steps would grow exponentially leading to a significantly larger state space. The time taken to search through this state space would also increase, necessitating partial state space generation and efficient search algorithms. To aid the verification of large systems modeled in CPN, CPN Tools is integrated with a model checking library called ASK-CTL [15]. Using computation tree logic (CTL) style syntax, temporal properties of the system can be specified. By exploiting strongly connected components (SCC) of the generated state space, useful system properties can be quickly checked by the tool, even for larger state spaces.

6. FUTURE WORK

For larger applications, with several timers and component interaction patterns, hand-writing the CPN token specification will prove to be cumbersome and error prone. Ongoing work will avoid this by integrating the temporal behavior specification for component-based applications onto the modeling framework used to generate deployment plans and infrastructure glue code, effectively closing the loop shown in Figure 1. The analysis model can be automatically generated by parsing through the structural properties of the design model, the component interactions, the assembly, and finally the temporal properties specified by the developer.

To make this model feasible for distributed applications or inherently parallel architectures, a *node_id* parameter can be added to the partition schedule and component executor thread data structures. The offset in the partition schedules managed on distributed nodes can be accounted for in the mf_off field. Thus, for nodes with no time-synchronization, the model will start the earliest schedule first, and all other schedules will be delayed by the initial offset. This model will also allow for multiple threads to be in the *Currently Running Thread* place as long as the running threads belong to different nodes.

The current analysis model expects the developer to provide

timing values for component operations as integers indicating the number of ticks each operation takes. A possible extension to this model would be to allow for a range of time values that the developer provides for each component operation, instead of a single value. To accommodate for probability-based temporal specifications, more appropriate petri net extensions need to be considered.

7. 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 inherent 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 inherent 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 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 simulation and state space search enables verification and validation of useful and necessary system properties, reducing development costs and increasing the reliability for such time-critical systems. We illustrate the utility of this approach with a few sample applications.

Acknowledgments: The DARPA System F6 Program 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.

8. REFERENCES

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

[2] X. Renault, F. Kordon, and J. Hugues, "Adapting models to model checkers, a case study : Analysing aadl using time or colored petri nets," in *Rapid System Prototyping, 2009. RSP '09. IEEE/IFIP International Symposium on*, pp. 26–33, June 2009.

[3] P. Martinez, J. Drake, and J. Medina, "Enabling model-driven schedulability analysis in the development of distributed component-based real-time applications," in *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, pp. 109–112, Aug 2009.

[4] J. L. Medina and A. G. Cuesta, "From composable design models to schedulability analysis with uml and the uml profile for marte," *SIGBED Rev.*, vol. 8, pp. 64–68, Mar. 2011.

[5] O. Sokolsky, I. Lee, and D. Clarke, "Schedulability analysis of aadl models," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 8 pp.–, April 2006.

[6] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee, "Incremental schedulability analysis of hierarchical real-time components," in *Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '06, (New York, NY, USA), pp. 272–281, ACM, 2006.

[7] K. Johnson, R. Calinescu, and S. Kikuchi, "An incremental verification framework for component-based software systems," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, (New York, NY, USA), pp. 33–42, ACM, 2013.

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

[9] T. L. et al., "Distributed real-time managed systems: A model-driven distributed secure information architecture platform for managed embedded systems," *IEEE Software*, vol. 31, no. 2, pp. 62–69, 2014.

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

[11] W. R. Otte, A. Dubey, S. Pradhan, P. Patil, A. Gokhale, G. Karsai, and J. Willemsen, "F6COM: A Component Model for Resource-Constrained and Dynamic Space-Based Computing Environment," in *Proceedings of the 16th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC '13)*, (Paderborn, Germany), June 2013.

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

[13] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, Apr 1989.

[14] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen, "Cpn tools for editing, simulating, and analysing coloured petri nets," in *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*, ICATPN'03, (Berlin, Heidelberg), pp. 450–462, Springer-Verlag, 2003.

[15] A. Cheng, S. Christensen, and K. H. Mortensen, "Model checking coloured petri nets exploiting strongly connected components," in *Proceedings of the International Workshop on Discrete Event Systems, WODES96. Institution of Electrical Engineers, Computing and Control Division*, pp. 169–177, 1997.