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

Pranav Srinivas Kumar, Abhishek Dubey and Gabor Karsai

Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37235, USA
{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. This paper describes how we can model and analyze applications for these systems in order to verify system properties such as lack of deadline violations. Our 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.

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. 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 comprehensive information architecture called **D**istributed **RE**al-time **M**anaged **S**ystem (DREMS) [1, 2] that addresses requirements for rapid component-based application development. In prior work, we have described the design-time modeling capability [3], and the component model used to build and execute applications [4]. The formal modeling and analysis method presented in this paper focuses on applications that rely on this foundational architecture.

The principle behind this 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. Using an appropriate model-based abstraction, the mapping from one domain to another remains valid under successive refinements in system development, including code generation. Application developers use domain-specific modeling languages to model the component assembly, component interactions, component execution code, operation sequencing, 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) [5] *analysis* model is generated. The analysis must ensure that, under the assumptions made about the components and the component architecture, the behavior of the system remains within the safe operational region. The results of this analysis will enable system refinement and re-design if required, before actual code development.

The remainder of this paper is organized as follows. Section 2 presents existing research relating to this paper; Section 3 provides a brief background on the DREMS Infrastructure and on the CPN formalism; Section 4 discusses the problem statement that is evaluated; Section 5 describes how this architecture is abstracted and modeled using CPN; Section 6 investigates the utility and scalability of state space analysis; Section 7 briefly describes how the analysis model is generated; Sections 8 and 9 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. Design models are transformed into relevant analysis models to study possible behaviors of the system and identify anomalies. When analyzing timing behavior, typically several exaggerated assumptions such as upper bounds on task execution times, service rates, maximum resource consumption etc are made. The results of system analysis using these assumptions are equally pessimistic. However, real-time systems with high criticality necessitate such assumptions to avoid the consequences of poor design.

Predictability of system behavior is achieved by obtaining upper bounds on 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 [6] and Timed Petri nets [7] to check for real-time properties such as deadline misses, buffer overflows etc. Similar to [7], our CPN-based analysis also makes use of observer places [8] that monitor the system behavior and look for real-time property violations and prompt completion of operations. However, [7] 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.

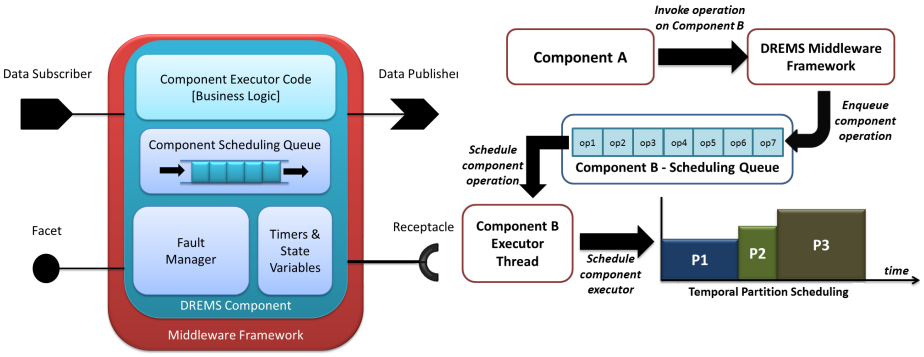
In the context of component-based systems, for complete real-time analysis, significant information must be obtained about the component assembly, the interaction patterns and the temporal behavior of components. The real-time model of the system is composed of real-time models of its constituent parts, each with its own temporal behavior. Using abstract model descriptors, [9] 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 [10] and analysis tools in the MAST environment, schedulability checks and priority assignment automation are performed. Note 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 [11] have been applied to 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, our analysis does not consider dynamic changes to component assembly at run-time, but it will be subject of future work.

3 Background

DREMS Components: Design and implementation of component-based software applications rests on the principle of assembly: *Complex systems are 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 of the individual components. Figure 1a shows the basic 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 can be invoked either



(a) DREMS Component (b) Component Operation Scheduling

Fig. 1: DREMS Infrastructure

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 invoke operations on other components using either RMI or AMI. A **publisher** port is a single point of data emission and a **subscriber** port is a single point of data consumption. Communication between publishers and subscribers is contingent on the compatibility of their associated topics (i.e. data types). More details on this component model can be found in [4].

Component Operation Scheduling: 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. As shown in Figure 1b, 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. Each operation is characterized by a priority and a deadline. The deadline here is the maximum acceptable time between the release of a component operation and the completion of that operation, measured starting from when the operation is enqueued onto the component's message queue. To facilitate component behavior that is free of deadlocks and race conditions, the component's execution is handled by a single thread. This single-threaded execution helps avoid synchronization primitives such as internal lock variables that lead to tenuous code development.

The DREMS OS scheduler enforces an ARINC-653 [12] style temporal and spatial partition scheme in order to schedule components grouped into processes. Temporal partitions, as shown in Figure 1b, are periodic fixed intervals of the processor time. Note that there are two levels of scheduling in DREMS: (1) Each component operation in the component-level is scheduled using a component-level scheduler, and (2) each component executor thread, on the system-level, is

scheduled by the OS in one of the temporal partitions, granting a slice of the CPU's time to those threads.

3.1 Colored Petri Nets

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. A transition can legally fire when all of its input places have necessary number of tokens, and when fires it produces tokens for its output places.

With Colored Petri nets (CPN) [5], tokens carry 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 the 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.

4 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 works by executing operations placed 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 of the sequence of computational steps of known duration that are executed inside each component operation, (2) knowledge of the worst-case estimated time taken by each computational step, and (3) the estimated worst-case time taken to initiate a remote function call and to process the 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 lies within the bounds laid out by the system specifications. Ideally, this is achieved by verifying such system properties as 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 describes the utility of an approach to identifying the subset of system behaviors that satisfy timing requirements and provide useful information to designers, e.g. partial thread execution orders.

5 Colored Petri net-based Modeling

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 [14] tool suite.

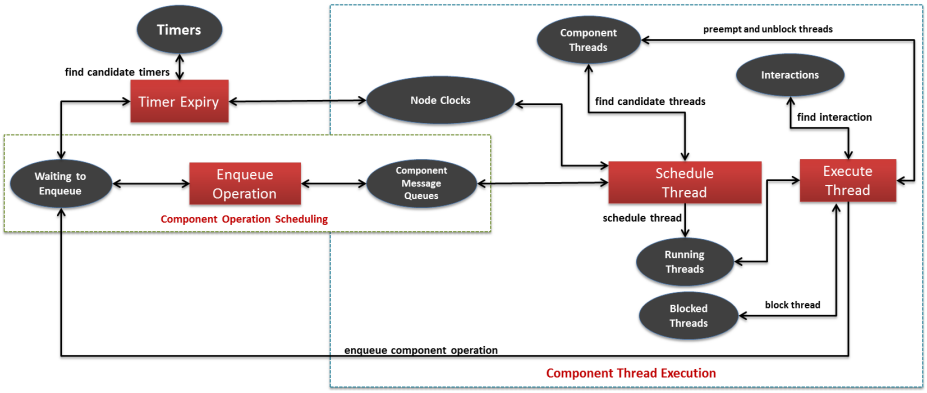


Fig. 2: Hierarchical CPN Analysis Model

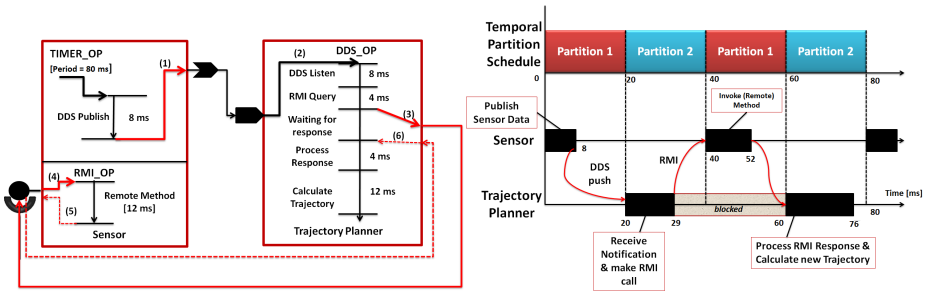
The CPN model captures the behavioral semantics of our component model described in [4], using knowledge of several factors that resolve the deployment of the component-based application. These factors include the following system properties: (1) configuration of temporal partition scheduling on each node of the distributed system, (2) location of each component being deployed (which temporal partition and which computing node) (3) properties of the component executor threads (thread priority), (4) properties of timers (period and offset), and (5) component interactions and assembly (i.e. the 'wiring').

Figure 2 shows a top-level structure of the CPN-based analysis model. The place *Component Threads* holds a token with a list of all executor threads responsible for component interactions. This list is maintained based on thread priorities on each node so that the highest priority ready thread is always chosen first by the OS scheduler. *Timers* maintains a list of all infrastructural timers in the application. All timer expiries at a specific clock value[1] are handled by the transition *Timer Expiry*. A timer can be used in our component model to trigger the execution of a component operation. DREMS components are dormant by default. Once initialized, a component executor is not eligible to run until there is an operation added to the component message queue. To start a sequence of component interactions, periodic or sporadic timers can be used to trigger an operation of a component.

If a timer triggers a component execution, this component is identified as a candidate for scheduling by *Schedule Thread*. This transition always schedules the highest priority thread that is ready to execute in the active partition on each node. If two threads of equal priority are eligible, the scheduler picks one at random and maintains a round-robin scheduling scheme. If the highest priority thread is not already servicing an operation request, the highest priority operation from its message queue is dequeued and scheduled for execution.

The *Component Message Queues* place is a list that manages the message queues of all components across all nodes. Every time a component thread exe-

[1] The clock values are integers.



(a) Component Assembly (b) Timing Diagram

Fig. 3: Trajectory Planning Application

cutes an operation, the completion of this operation could trigger another component into execution. For instance - the completion of an RMI query on a client component triggers a server-side RMI operation that this server will have to execute. Such interactions are derived from the modeling tools and appropriate tokens are generated in place *Interactions*. When executing component threads, *Execute Thread* checks to see if the execution has any effect on the running thread or on other threads. Therefore, when the client thread completes an RMI query, this thread is moved to *Blocked Threads* and a server RMI operation is placed in *Waiting to Enqueue*. Later, when the server thread is scheduled, the client is unblocked appropriately.

6 State Space Analysis

Given a CPN model (that was generated from a component architecture and deployment model), a state-space of the system can be constructed using the semantics of CPN. This state space is infinite, however, in practice, it is often sufficient to consider some finite subset, starting from a initial state up to a few hyperperiods of the partition scheduler. In order to describe the utility of state space analysis, we consider a simple trajectory planning application (TPA). The component assembly for this application is shown in Figure 3a. A Sensor component periodically publishes on a trigger topic, notifying the Trajectory Planner of the existence of new sensor data. Once the notification is received, the Trajectory Planner makes an RMI call to retrieve the data structure of sensor values, using which the satellite trajectory is updated. The sequence of steps in each of these operations is referred to as the business logic of the operation. This business logic is modeled using a textual language in the modeling tools, in which the designer specifies the macro execution steps in a component operation along with worst-case estimated time taken on each step. Figure 3b shows the expected timing diagram.

The analyzable states of this system are observed in the markings of the various CPN places in the model. Using the built-in state space analysis in *CPN Tools* a bounded state space of the system is generated. Using both standard

and user-defined queries, this state space is searched to check system properties like lack deadline violations and deadlocks, bounds on response times etc.

Deadline Violation Detection: Each time a component operation is scheduled, the clock value of the node is recorded as the "start time" of the operation. If this operation is incomplete when the clock reaches the operation's deadline, a deadline violation is detected. Using the *SearchNodes* function in CPN Tools, the deadline violations on any component operation can be identified by observing all component operations each time the node-specific clock progresses. In Figure 3b, the *DDS_OP* on the Trajectory Planner takes 56 ms to complete, measured from when the operation was enqueued and marked as ready. If the deadline of this operation is set to 50 ms, a state space search would reveal a deadline violation when the clock reaches 51 ms.

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

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

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 large systems with groups of applications and increased concurrency, an equally large state space is required to observe the tree of possible thread executions and operational behaviors. This analysis model has been identified to scale well for medium-sized applications, tested up to 100 components distributed on up to 5 computing nodes. Table 1 summarizes these results.

Table 1: Scalability Results

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

7 Analysis Model Generation

As mentioned in Section 5, the control flow and timing details of component operations are directly integrated into the design-time modeling framework. Using the formal domain-specific model of the system, the configuration of the partition scheduling and component assembly are derived and translated into meaningful CPN tokens. The business logic of each component operation is expressed using a textual language with one attribute per interaction per instance of each component being deployed. Model interpreters parse through this complete design model, instantiating CPN model templates and combining these instances to generate a single integrated .cpn file to analyze the entire system.

8 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, for component operation scheduling; and novel interaction patterns (e.g. reliable broadcast). Also, 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.

9 Conclusions

Distributed real-time systems operating in dynamic environments, and running mission-critical applications face strict timing requirements to operate safely. This paper presents a Colored Petri Net-based approach to capture the architecture and temporal behavior of such 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.

Acknowledgments

The DARPA System F6 Program and the National Science Foundation (CNS-1035655) supported this work. Any opinions, findings, and conclusions or rec-

ommendations expressed in this material are those of the authors and do not reflect the views of DARPA or NSF.

References

1. Dubey, A., Emfinger, W., Gokhale, A., Karsai, G., Otte, W., Parsons, J., Szabo, C., Coglio, A., Smith, E., Bose, P.: A Software Platform for Fractionated Spacecraft. In: Proceedings of the IEEE Aerospace Conference, 2012, Big Sky, MT, USA, IEEE (2012) 1–20
2. et al., T.L.: Distributed real-time managed systems: A model-driven distributed secure information architecture platform for managed embedded systems. IEEE Software **31** (2014) 62–69
3. Dubey, A., Gokhale, A., Karsai, G., Otte, W., Willemsen, J.: 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 (2013)
4. Otte, W.R., Dubey, A., Pradhan, S., Patil, P., Gokhale, A., Karsai, G., Willemsen, J.: 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 (2013)
5. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer (2009)
6. Renault, X., Kordon, F., Hugues, J.: From aadl architectural models to petri nets: Checking model viability. In: Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC '09. IEEE International Symposium on. (2009) 313–320
7. Renault, X., Kordon, F., Hugues, J.: 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. (2009) 26–33
8. Alpern, B., Schneider, F.B.: Verifying temporal properties without temporal logic. ACM Trans. Program. Lang. Syst. **11** (1989) 147–167
9. Lopez, P., Medina, J., Drake, J.: Real-time modelling of distributed component-based applications. In: Software Engineering and Advanced Applications, 2006. SEAA '06. 32nd EUROMICRO Conference on. (2006) 92–99
10. Harbour, M.G., Garcia, J.J.G., Gutierrez, J.C.P., Moyano, J.M.D.: Mast: Modeling and analysis suite for real time applications. In: In 13th Euromicro Conference on Real-Time Systems. (2001) 125
11. Johnson, K., Calinescu, R., Kikuchi, S.: 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, ACM (2013) 33–42
12. ARINC Incorporated Annapolis, Maryland, USA: Document No. 653: Avionics Application Software Standard Inteface (Draft 15). (1997)
13. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77** (1989) 541–580
14. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K.: 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, Springer-Verlag (2003) 450–462