Experimental Validation of Timing Analysis for Component-based Distributed Real-time Embedded Systems

Pranav Srinivas Kumar and Gabor Karsai
Institute for Software-Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37235, USA
Email:{pkumar, gabor}@isis.vanderbilt.edu

***Abstract*—In prior work [1] [2], we have presented a Colored Petri net (CPN) based approach to modeling and scalable timing analysis of component-based distributed real-time systems. We have also presented improvements to our analysis methods, specifically through various structural and state-space reduction techniques that make the model more scalable, and open to extensions. In this paper, we present experimental validation of this timing analysis, presenting our evaluation workflow, measured execution times on component operations compared against our timing analysis results. The experiments cover various component interaction patterns, mixed-criticality thread execution, and distributed scenarios facilitated by our Resilient Cyber-Physical Systems (RCPS) testbed [3]. Results show the correctness of our CPN approach and the closeness of its predictions in composed component assemblies.**

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

I. Introduction

Developing efficient and reliable component-based software for large-scale distributed embedded systems is difficult. The software performance is often coupled with low-power hardware and sometimes interacting with an external physical environment. Using component-based software has not only improved the maintainability of large complex code bases but also enabled structured analysis methods that rely on the principle of *compositionality* [4]: properties of a composed system can be derived from the properties of its components and connections. In prior work [1], we have considered one specific component model, DREMS [5], and developed a Colored Petri net-based timing analysis approach to model the structural and behavioral semantics of DREMS components, and presented various state space analysis techniques that can be applied to verify and analyze the timing properties of a composed system e.g. lack of deadline violations deadlocks etc. More recently, we have also presented [2] analysis improvements by leveraging structural reduction techniques and advanced state space generation methods that make our analysis model more scalable and relevant for industrial scale scenarios.

Experimentally validating our timing analysis results is an important and necessary requirement. In order to obtain any level of confidence in our CPN-based work, the system design model needs to be completely implemented, and deployed on the target hardware platform. We have constructed an experimental embedded systems testbed [3] to simulate and analyze resilient cyber-physical systems - consisting of 32 Beaglebone Black development boards [6]. We have chosen the light-weight ROS [7] middleware layer and implemented a DREMS-style component model, ROSMOD [8] on top of it. Our goal with this work is to (1) develop a set of distributed component-based applications, (2) translate this design model to our CPN analysis model, (3) deploy these applications on our testbed and accurately measure operation execution times, and finally (4) perform state space analysis on the generated CPN model to check for conservative results, compared against the real system execution.

Our contributions in this paper are as follows:

1) We present our evaluation workflow, including our hardware testbed, software deployment framework, and various real-time system properties that were enforced.
2) We present a few component assemblies and interaction patterns, including the integration of long running operations, executed on our RCPS testbed and discuss operational performance with execution time plots.
3) We describe how hardware-dependent metrics such as worst-case execution times propagate into our Colored Petri net-based analysis model. This is required in order to establish any level of consistency between the real system and its theoretical execution model.
4) We present representative execution plots derived from our CPN, specifically plots of the worst execution trace realized during state space analysis. Comparing the real-world execution plots against our CPN analysis plots shows that our analysis results are conservative but close approximations of the real system execution.

The remainder of this paper is organized as follows. Section II presents related research, reviewing and comparing existing analysis tools and formal methods. Section III presents our experimental testbed, briefly describing the architecture and evaluation workflow. Section IV describes the operation

execution semantics for our component model, presenting the need for our timing analysis methods. Section V briefly summarizes our Colored Petri net-based analysis model in order to establish the level of refinement involved. Section VI describes our experimental evaluation, presenting execution time plots of different interaction patterns and component assemblies. Finally Sections VII and VIII briefly mention our planned future work and concluding remarks.

II. Related Research

Petri nets enable the modeling and visualization of dynamic system behaviors that include concurrency, synchronization and resource sharing. Theoretical results and applications concerning Petri nets are plentiful [9], [10], especially in the modeling and analysis of discrete event-driven systems. Models of such systems an be either *untimed* or *timed* models. Untimed models are those approximations where the order of the observed events are relevant to the system but the exact time instances when a state transitions is not considered. Timed models, however, study systems where its proper functioning relies on the time intervals between observed events. Petri nets and extensions have been effectively used for modeling both untimed [10] and timed systems [11]. For a detailed study of Petri nets and its applications, the reader is referred to standard textbooks [12], [13] and survey papers [14], [15], [16].

Petri nets have evolved through several generations from low-level Petri nets for control systems [13] to high-level Petri nets for modeling dynamic systems [17] to hierarchical and object-oriented Petri net structures [18] that support class hierarchies and subnet reuse. Several extensions to Petri nets exist, depending on the system model and the relevant properties being studied e.g. Timed Petri nets [19], Stochastic Petri nets [20], [21] etc. High-level Petri nets are a powerful modeling formalism for concurrent systems and have been widely accepted and integrated into many modeling tool suites for system design, analysis and verification.

Teams of researchers have, in the past, identified the need for in-depth timing analysis tools that integrate with complex system design challenges, especially in model-driven architectures [22]. Development tools like MARTE [23] and AADL [24] provide a high-level formalism to describe a DRE system, at both the functional and non-functional level. MARTE (Modeling and Analysis of Real-time Embedded Systems) defines the foundations for model-based description of real-time and embedded systems. MARTE is also concerned with model-based analysis and integration with design models. The intent here is not to define new techniques to analyze real-time systems, but instead to support them. So, MARTE supports the annotation of models with information required to perform specific types of analysis such as performance and schedulability analysis. However, the framework is more generic and intended to refine design models to best fit any required kind of analysis. Although tools exist that exercise common schedulability analysis methods like Rate Monotic Scheduling (RMA) analysis, there are very few usable tools that address the complex challenge of testing and certifying behaviors of complete, composed systems.

In general MARTE provides a generic canvas to describe and analyze systems. The user is required to add domain-specific and system-specific properties and artifacts on top of the generic platform. Compared to MARTE, AADL (Architecture Analysis and Design Language) comes with a standalone and complete semantics that is enforced by the standard. In [22], the authors propose a bridge that translates AADL specifications of real-time systems to Petri nets for timing analysis. This formal notation is deemed to be well-suited to describe and analyze concurrent systems and provides a strong foundation for formal analysis [25] methods such as structural analysis and model checking. The high-level goal is to check and verify AADL models for properties like deadlock-freedom and boundedness. The workflow presented here is similar to the proposed work in this thesis in the sense that a system design model along with user-specified properties are translated into a high-level Petri net-based analysis model.

However, there are some potential improvements to this work. Not only is the generated Petri net structure hard to follow, it is seemingly composed of sub-Petri nets, one for each thread (and its lifecycle) in each process. It is clear that although the transformation is sound, the generated Petri net models are going to be intractably large for complicated scenarios. The state space of the Petri net is dependent on the number of places in the net and the corresponding internal states. The generated net would not scale well for large process sets or distributed scenarios without using state space reduction techniques that rely on symmetry [26]. Such troubles can be alleviated by using a high-level Petri net such as Colored Petri nets where much more information can be packed in a Petri net token. Complex token data structures reduce the number of places required to describe a system model e.g. a list of C-style struct data structures can abstractly model a set of processors. This reduces the number of places that would be required to represent a full system. Such modeling constructs are essential in component-based systems where the full system is typically a large assembly of tested black box components. Lastly, the modeling constructs used are strictly bound to AADL and cannot be easily modified for systems not modeled using AADL, especially strictly-defined component models with precise execution semantics.

III. Experimental Setup

Cyber-Physical Systems require design-time testing and analysis before deployment. Several CPS scenarios require strict safety certification due to the mission-critical nature of the operation, e.g. flight control and automation. It is often times impossible to test control algorithms, fault tolerance procedures etc. on the real system due to both cost and hardware accessibility issues. To counter these issues, there are two principle methods in which a CPS can be tested and analyzed: (1) Construct a complete model of the CPS in a simulation environment e.g. Simulink [27] and simulate the system while accounting for run-time scenarios, (2) Establish

a testing environment that can closely resemble the real CPS in both hardware and software. The problem with simulations is that it is hard to establish the network topology, emulate the application network and base processing power while running a physics simulation in the loop. Our RCPS testbed, as shown in Figure 1, implements the latter alternative. For further details on the architecture of this testbed, readers are referred to our earlier work [3].

Fig. 1: RCPS Testbed

Using the RCPS testbed, a wide variety of DRE use-cases can be tested and accurately measured. This section briefly considers a few samples that test the various interaction patterns of DREMS, that are available to application developers and compares the measured results against the predicted worst-case execution time profiles of the modeled applications in our Colored Petri net-based analysis model. The expected result in all of these cases is to observe close but pessimistic behavior simulation from our analysis model i.e., the CPN analysis should be able to simulate and analyze the behaviors of the test cases and provide comparable and close approximations of the execution time plots while ensuring that the predicted worst-case execution times, response times, processor utilization etc. are always conservative. This means that the real deployment of the application threads *never* exceeds the predicted worst-case execution times. Such results will validate the predicted timing properties of the system and ensure that the CPN analysis is useful and reliable.

IV. Component Execution Semantics

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. Application developers provide the functional, *business-logic* code that implements operations on the state variables e.g. a PID control operation could receive the current state of dynamic variables from a *Sensor* Component, and using the relevant gains, calculate a new state to which an *Actuator* component should progress the system. 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.

To facilitate component behavior that is free of deadlocks and race conditions, the component's execution is handled by 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 by a single component-specific executor thread.

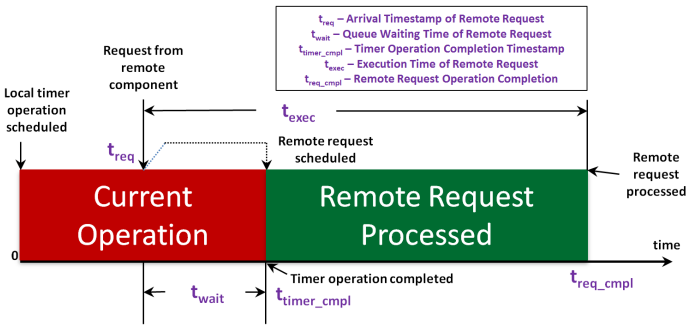

Fig. 2: Component Operation - Execution Semantics

Figure 2 shows the execution semantics of a component operation executed on a lone component executor thread. A simplifying assumption to describe the semantics is that this component is the only component thread executing on this CPU. Assume that at $t = 0$, this component is processing the expiry of a local timer. This operation is expected to complete at $t = t_{timer\_cmpl}$. However, at $t = t_{req}$, a service request is received from some remote component. Since the component operation scheduling is non-preemptive, regardless of the priority of this service, the request is not processed until t_{next}. Therefore, the request is waiting in the message queue for $t_{wait} = t_{timer\_cmpl} - t_{req}$. At $t = t_{timer\_cmpl}$, the timer operation is marked as complete and the service request is processed. The total execution time of this service operation is calculated including the duration of the time for which this request waited in the component message queue i.e. $t_{exec} = t_{req\_cmpl} - t_{req}$. The wait times in the queue are further worsened when OS scheduling non-determinism is taken into account. There are specifically three ways in which the OS scheduling can delay operations: (1) if the application process

is concurrently executing multiple component threads, then the threads are scheduled in Round-Robin fashion by the OS, (2) when mixed-criticality application processes are scheduled in tandem, the OS uses fixed-priority Round-Robin scheduling to schedule the process threads, and finally (3) temporally partitioned OS schedules further cause delays on component thread scheduling, which directly affects the scheduling and timely completion of component operations.

V. CPN Analysis Model

The CPN analysis model, as modeled in CPN Tools [28], is shown in Figure 3. *Places*, shown as ovals, in this model contain colored (typed) tokens that represent the state of interest for analysis e.g. *Clocks* place holds tokens of type *clocks* maintaining information regarding the state of the clock values and temporal partition schedule on all computing nodes. *Transitions*, shown as rectangular boxes, are responsible for executing this model, progressing the state of the modeled system and transferring tokens between places. *Arcs*, between transitions and places dictate the token flow and data structure manipulations. All arcs contain emphinscriptions, which are essentially function calls, written in Standard ML, that manipulate token structures e.g. arc inscriptions in the arc from the transition *Timer_Expiry* to the place *Timers*, manipulate all timer tokens by updating the timer expiry offsets.

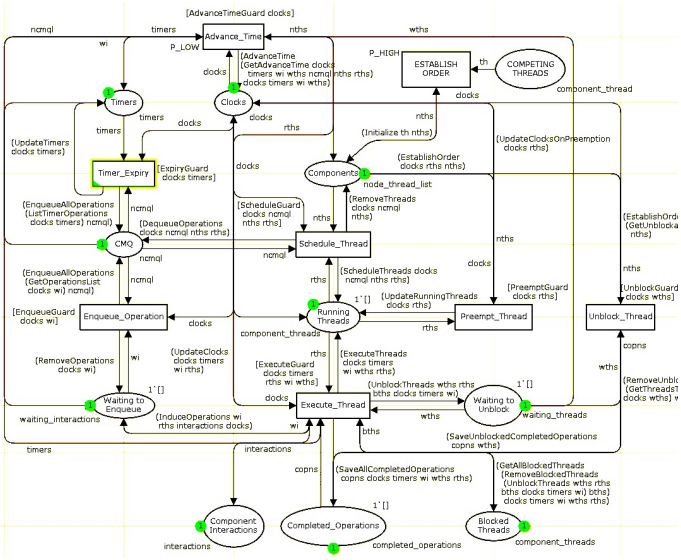

Fig. 3: Complete CPN Timing Analysis Model

From the design model of the system, we generate the initial CPN tokens that are injected into places in this analysis model. Using the in-built state space analysis engine, we analyze the state space of the parameterized model to compute useful system properties e.g. processor utilization, execution time plots, deadline violations etc. The modeling concepts in Figure 3 can be divided and categorized based on system-level concepts being analyzed. Figure 4 shows the organizational structure of this CPN. Below, we describe each of these structural divisions in detail.

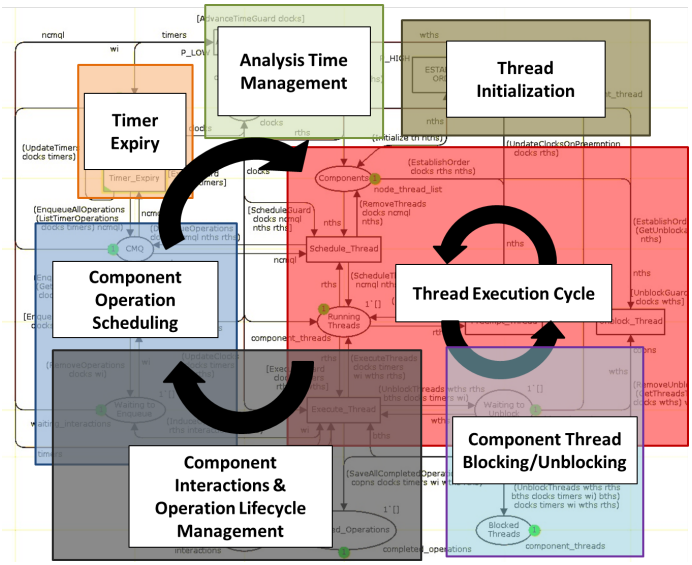

Fig. 4: Structural Aspects

VI. Experimental Evaluation

Experimental validation requires that online measurements of the real-time system match with the timing analysis results in a way that the timing analysis results are always close but conservative. If the timing analysis results predict a deadline violation, this does not necessarily mean that the real system will violate deadlines but if the timing analysis and verification guarantees a lack of deadline violations, then the real system should follow this prediction. One of the biggest assumptions in our CPN work is the knowledge of worst-case execution times of the individual steps in the component operations. We have previously designed [2] a business-logic modeling grammar that captures the temporal behavior of component operations, especially WCET metrics for the different code blocks inside an operation. For example, consider a simple client-server example as shown in Figure 5. The client component is periodically triggered by an internal timer and executes a synchronous remote method invocation to a remote server component. The interaction here demands that the client component be blocked for the duration of time it takes the server to receive the operation, process its message queue, execute the relevant callback, and respond with output.

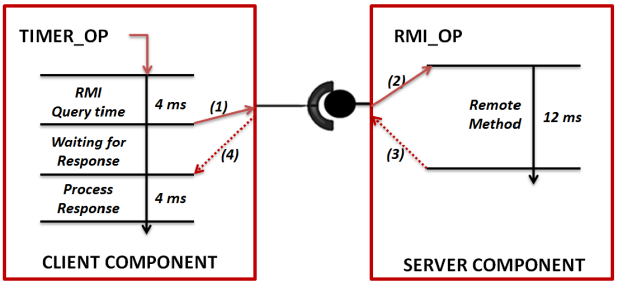

Fig. 5: RMI Application

Note that in the above figure, we only annotate isolated

code blocks that take a fixed amount of execution time under a specific hardware architecture. These are the only measurements that we can reliably make with repeated testing and instrumentation. The client-side blocking delays is not measured, but instead realized through state space analysis because this delay is dependent on the state of the server's host.

WCET of component operational steps needs to be measured by having the component operation execute at real-time priority with no other component threads intervening this process. This measurement gives us a *pure execution time* of the code block. The process must be repeated for all component operations to obtain meaningful worst-case estimates that are tailored to the target platform. Obtaining the WCET values by this method is not only more realistic but also an accurate representation of the target system. Once these individual numbers are obtained, the values are plugged into the CPN through our business-logic models.

The remainder of this section presents various primitive interaction patterns and assemblies. The results are restricted to simple cases, though we have tested on medium-to-large scale examples spanning 25-30 computing nodes, and with 100 components. As mentioned earlier, in all of our tests, we use the ROS [7] middleware and our ROSMOD [8] component model.

A. Client-Server Interactions

As shown in Figure 5, a simple client server example involves a periodically triggered client component that fetches data from a remote server. Figure 6 shows our experimental observation of a simple distributed client-server sample. The client is triggered every 500 ms, and performs floating-point calculations in a loop. The loop bound is a random variable having a uniform distribution between some peak iteration count and 60% of this peak. The server periodically receives this operation request and responds to it, taking about 1.2s to complete each operation instance.

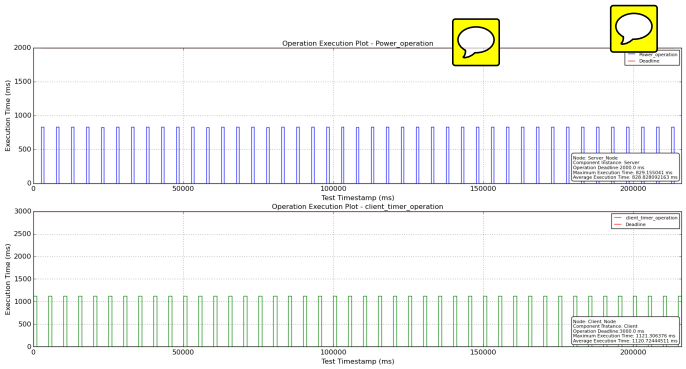

Fig. 6: Experimental Observation: Client-Server Interactions

Figure 7 shows the execution time plot derived from our CPN. As expected, since there are no other interruptions on the server side, the server is able to promptly respond to the client.

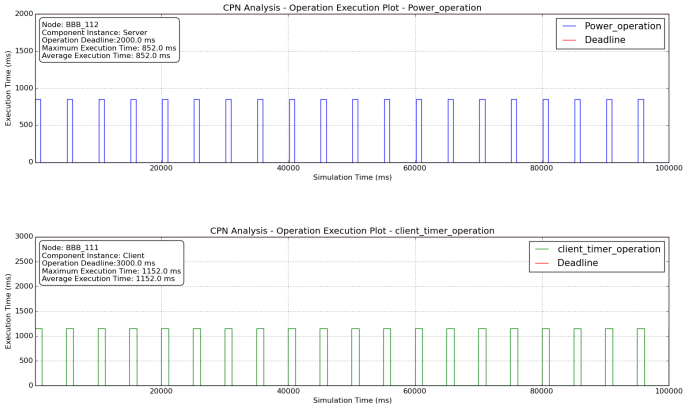

Fig. 7: CPN Analysis Results: Client-Server Interactions

B. Publish-Subscribe Interactions

Similar to the earlier example, consider a simple anonymous publish-subscribe interaction. A publisher is periodically triggered by a timer when this component broadcasts a message on a topic. A subscribing component receives this message and performs some computation.

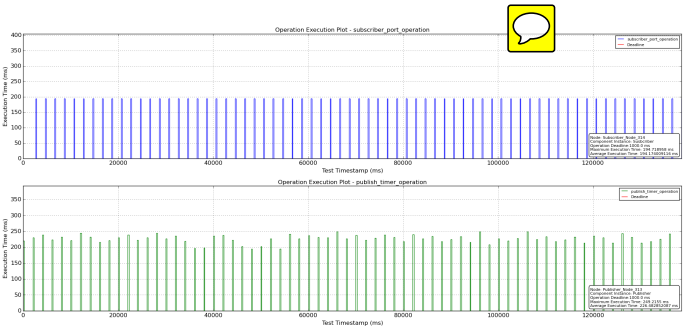

Fig. 8: Experimental Observation: Publish-Subscribe Interactions

Figure 8 shows our testbed observations and Figure 9 shows our CPN analysis results. As evident, the CPN results closely match and validate this sample.

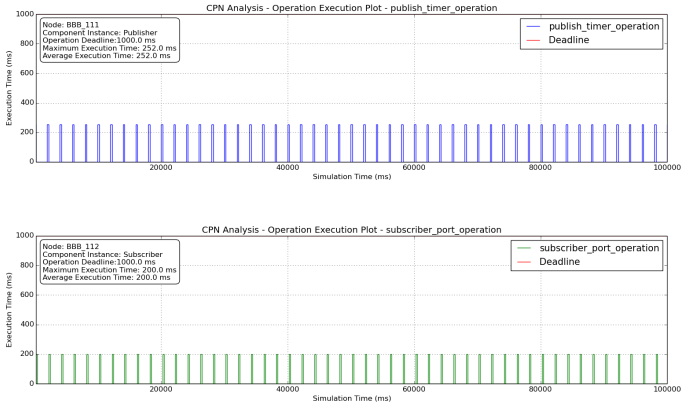

Fig. 9: CPN Analysis Results: Publish-Subscribe Interactions

C. Trajectory Planner

In the past [1], we used a *Trajectory Planner* deployment to illustrate the utility of our state space analysis. Figure 10 shows the execution time plot of this sample. A Sensor component is periodically triggered by a timer at which point it publishes a notification to the Trajectory Planner, alerting the planner of new sensor state. The planner component, on receiving this message, queries a remote server when ready and obtains the sensor state information.

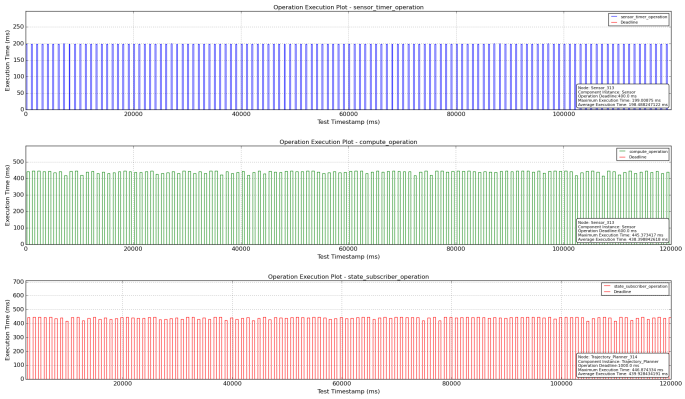

Fig. 10: Experimental Observation: Trajectory Planner

This is a common interaction pattern in Cyber-Physical systems since embedded sensors are updated at a much higher frequency than a path planning entity. Thus, the planner can query the sensor at a lower rate to sample the sensor state. In this example, the planner is matching the frequency of the sensor since the execution cost is low. However, when more components are added to this deployment, the planner would have to fetch sensor state less frequently so as to not affect other system-level deadlines.

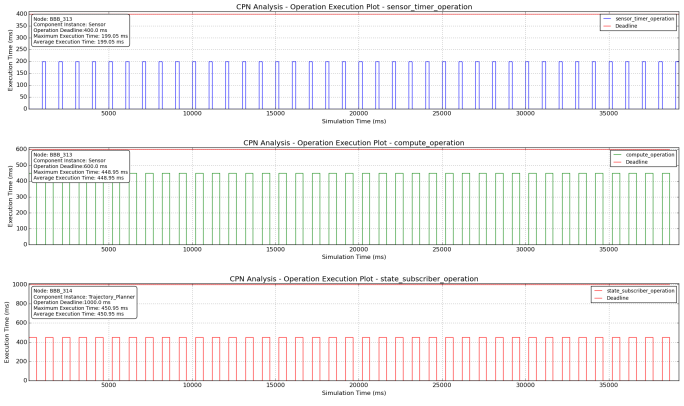

Fig. 11: CPN Analysis Results: Trajectory Planner

D. Time-triggered Operations

Time-triggered operations are an integral part of our component model. DREMS components are dormant by default. A timer has to trigger a inactive component for all subsequent interactions to happen. Since the DREMS component model supports various scheduling schemes on a single component message queue, this following test evaluates a priority first-in first-out (PFIFO) scheme. Multiple timers are created in a single component, each with a unique priority and period. A timer with a high frequency is assigned a high priority. Figure 12 shows our experimental observations on a 5-timer example.

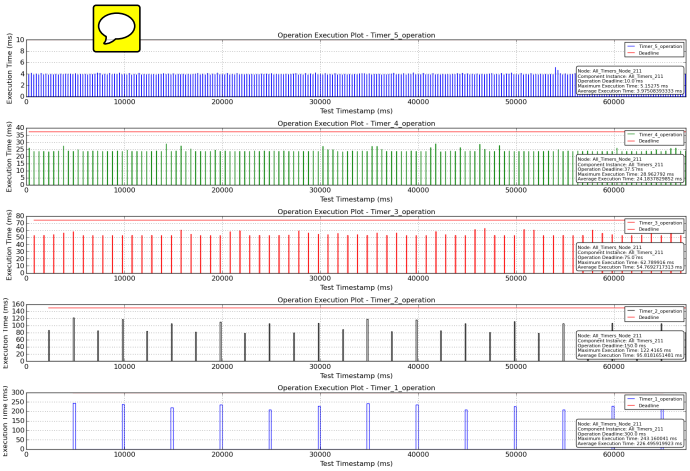

Fig. 12: Experimental Observation: Periodic Timers

Since DREMS components are associated with a single executor thread and component operations are also non-preemptive, a low-priority operation could theoretically run forever, starving a higher priority operation from ever executing, leading to deadline violations e.g. *Timer_1_operation* can affect all other higher priority timers. Figure 13 shows our CPN prediction where such a scenario is evident. It can be seen that *Timer_5_operation*, the timer with the highest priority is periodically seeing spikes in execution time, courtesy of other lower priority operations consuming CPU without preemption.

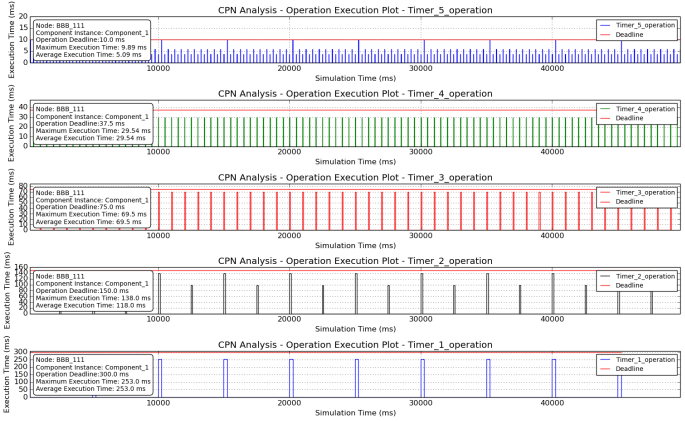

Fig. 13: CPN Analysis Results: Periodic Timers

It must be noted here that the execution time values assigned to each timer operation in our CPN is the pure execution time i.e. the time taken for each timer operation to execute on the target CPU without interruption. This is the case for all operational execution times injected into the analysis model. If this is not done, then due to scheduling delays and interaction patterns, the CPN results will become gross overestimates.

E. Long-Running Operations

Our DREMS component model implements a non-preemptive component operation scheduling scheme. A component operation that is in the queue, regardless of its priority, must wait for the currently executing operation to run to completion. This is a strict rule for operation scheduling and does not work best in all system designs e.g. in a long-running computation-intensive application, rejuvenating the executing operation periodically and restarting it at a previous checkpoint increases the likelihood of successfully completing the application execution. In applications executing long-running artificial intelligence (AI) search algorithms e.g. flight path planning algorithms, the computation should not hinder the prompt response requirements of highly critical operation requests such as sudden maneuver changes. Our DREMS component model does not support the *cancellation* of long-running component operations to service other highly critical operations waiting in the queue. With a few minor modifications to our scheduling schemes, long running operations can, however, be suspended if a higher priority waiting operation requires service. With these additions, we are able to model and analyze component-based systems that support long-running operations, with checkpoints, enabling the novel integration of AI-type algorithms into our design and analysis framework.

1) Challenges: One of the primary challenges here is to identify the semantics of a long-running component operation i.e. the scenarios under which the component operations scheduler suspends a long-running operation in favor of some other operation waiting in the queue. If a long-running computation is modeled as a sequence of execution steps with equally-spaced checkpoints, then the operation would execute one step at a time and the scheduler would wait for the nearest checkpoint to reach, before suspending the operation (if needed). An important challenge here is accurately identifying the priority difference between the long-running operation and the waiting operation. If the long-running operation is one checkpoint away from completion e.g. 100-200 ms of execution time, then strictly following our suspension rules would not be the most prudent choice since this operation is almost complete. However, if the waiting operation is a critical one, then regardless of the state of the long-running operation, the executing operation must be suspended. Secondly, the modeled long-running computation semantics must be incorporated into our component model so that any analysis results obtained can be suitably validated.

2) Implementation and Results: In each long-running operation, we, therefore, include a synchronous *checkpoint step*. The only assumption we make about this long-running operation is the periodicity of these checkpoint steps i.e. we know how frequently a new checkpoint is reached and we assume that the search algorithm used by the long-running operation is capable of reaching a safe state (the checkpoint) before suspending itself if required. If a higher priority operation is ready and waiting in the queue, the long-running operation runs till the next checkpoint is reached, then suspends. The higher priority operation is then processed. Figure 14 shows the *Software Model* for a component assembly with long running operations.

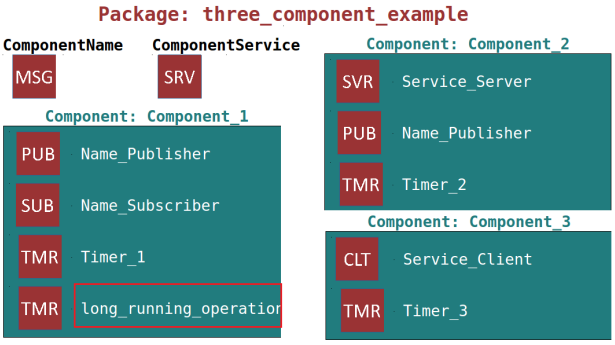

Fig. 14: Long Running Operation - Software Model

The assembly consists of three components. Components *Component_1* and *Component_2* periodically publish on the *ComponentName* message. *Component_3* periodically queries the server in *Component_2*. During these interactions, *Component_1* is performing a long running operation, the duration of which, is magnitudes larger than the average execution time of all other operations. Figure 15 shows the execution time plot of this scenario, as measured on our testbed.

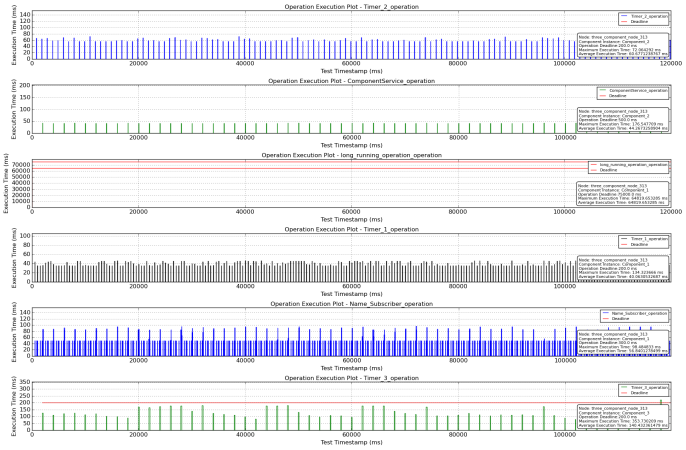

Fig. 15: Experimental Observation: Composed Component Assembly

For the CPN analysis, in order to obtain pure execution times of all these operations, each operation on each component is executed as a stand-alone function on the hardware. This way, we know the average and worst-case execution times of all operational steps with minimal interruptions. These numbers are injected into our generated CPN and state space analysis is performed. Figure 16 shows our CPN analysis results for the same assembly.

VII. Future Work

All of the results presented in this paper make an important assumption about the network - the network resources available to each component is much larger than the requirements

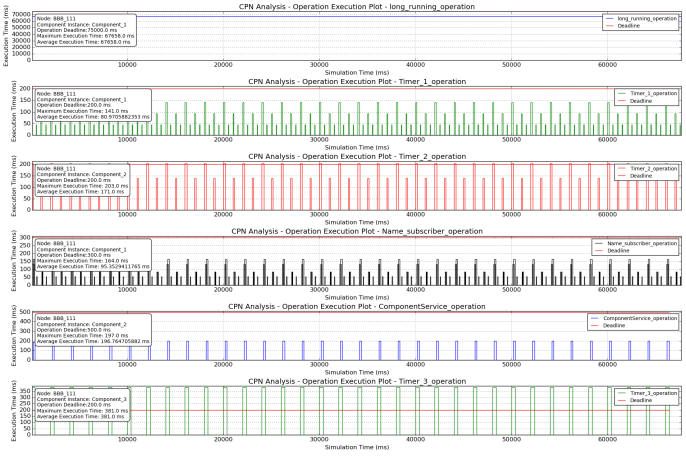

Fig. 16: CPN Analysis Results: Composed Component Assembly

of the application i.e. there is no buffering delays on the network queues when components periodically produce data. We are currently working on integrating established Network Calculus-based analysis methods [29] into our CPN analysis for a more accurate profile of the analyzed application. This way, we can also model the system network profile (available bandwidth as a function of time) and realize the application data production profile, accounting for network delays caused by buffering.

VIII. Conclusions

In this paper, we have presented experimental validation for our Colored Petri net-based timing analysis methods for component-based distributed real-time systems. The validation covers a variety of component assemblies, interaction patterns and concepts, including publish subscribe-style messaging, client server-style querying, time-triggered interactions and long running operations. The results show close but conservative estimates from state space analysis and validate the utility of our tools and methods.

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.

References

[1] P. S. Kumar, A. Dubey, and G. Karsai, "Colored petri net-based modeling and formal analysis of component-based applications," in *11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2014*, 2014, p. 79.

[2] P. Kumar and G. Karsai, "Integrated analysis of temporal behavior of component-based distributed real-time embedded systems," in *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2015 IEEE International Symposium on Real-time Computing (ISORC)*, April 2015, pp. 50–57.

[3] P. Kumar, W. Emfinger, and G. Karsai, "Testbed to simulate and analyze resilient cyber-physical systems," in *Rapid System Prototyping, 2015. RSP '15.*, October 2015.

[4] H. Jifeng, X. Li, and Z. Liu, "Component-based software engineering," in *Theoretical Aspects of Computing–ICTAC 2005*. Springer, 2005, pp. 70–95.

[5] 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.

[6] "Beaglebone Black," http://beagleboard.org/BLACK/.

[7] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[8] P. Kumar, W. Emfinger, A. Kulkarni, G. Karsai, D. Watkins, B. Gasser, C. Ridgewell, and A. Anilkumar, "Rosmod: A toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ros." in *Rapid System Prototyping, 2015. RSP '15.*, October 2015.

[9] R. David and H. Alla, "Petri nets for modeling of dynamic systems: A survey," *Automatica*, vol. 30, no. 2, pp. 175–202, 1994.

[10] L. E. Holloway, B. H. Krogh, and A. Giua, "A survey of petri net methods for controlled discrete event systems," *Discrete Event Dynamic Systems*, vol. 7, no. 2, pp. 151–190, 1997.

[11] W. Zuberek, "Timed petri nets definitions, properties, and applications," *Microelectronics Reliability*, vol. 31, no. 4, pp. 627–644, 1991.

[12] J. L. Peterson, "Petri nets," *ACM Computing Surveys (CSUR)*, vol. 9, no. 3, pp. 223–252, 1977.

[13] W. Reisig, *Petri nets: an introduction*. Springer Science & Business Media, 2012, vol. 4.

[14] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[15] M. Zhou and K. Venkatesh, *Modeling, simulation, and control of flexible manufacturing systems: a Petri net approach*. World Scientific, 1999, vol. 6.

[16] R. Zurawski and M. Zhou, "Petri nets and industrial applications: A tutorial," *Industrial Electronics, IEEE Transactions on*, vol. 41, no. 6, pp. 567–583, 1994.

[17] K. Jensen and G. Rozenberg, *High-level Petri nets: theory and application*. Springer Science & Business Media, 2012.

[18] G. A. A. F. De Cindio and G. Rozenberg, "Object-oriented programming and petri nets," 2001.

[19] J. Wang, *Timed Petri nets: Theory and application*. Springer Science & Business Media, 2012, vol. 9.

[20] F. Bause and P. S. Kritzinger, *Stochastic Petri Nets*. Springer, 1996.

[21] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modelling with generalized stochastic Petri nets*. John Wiley & Sons, Inc., 1994.

[22] X. Renault, F. Kordon, and J. Hugues, "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*, March 2009, pp. 313–320.

[23] *UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)*, OMG Document realtime/05-02-06 ed., Object Management Group, May 2005.

[24] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The Architecture Analysis & Design Language (AADL): An Introduction," DTIC Document, Tech. Rep. ADA455842, 2006.

[25] C. Girault and R. Valk, *Petri nets for systems engineering: a guide to modeling, verification, and applications*. Springer Science & Business Media, 2013.

[26] A. P. Sistla and P. Godefroid, "Symmetry and reduced symmetry in model checking," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 26, no. 4, pp. 702–734, 2004.

[27] "Simulink," http://www.mathworks.com/products/simulink/.

[28] 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*, ser. ICATPN'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 450–462. [Online]. Available: http://dl.acm.org/citation.cfm?id=1760066.1760097

[29] W. Emfinger, G. Karsai, A. Dubey, and A. Gokhale, "Analysis, verification, and management toolsuite for cyber-physical applications on time-varying networks," in *Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, ser. CyPhy '14. New York, NY, USA: ACM, 2014, pp. 44–47. [Online]. Available: http://doi.acm.org/10.1145/2593458.2593459