

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—Safety-critical *Distributed Real-time Embedded* (DRE) systems are characterized by strict timing requirements and resilient operational demands. Such systems require comprehensive design-time modeling and analysis techniques to ensure predictable and dependable runtime behaviors. In prior work, we have presented a *Colored Petri net* (CPN) based approach to modeling and scalable timing analysis of component-based DRE systems. Component-based design models are translated into scalable CPN-based timing analysis models, capable of evaluating the system and deriving useful timing properties such as lack of deadlocks, deadline violations etc. 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, deployed, and accurately measured on our *Resilient Cyber-Physical Systems* (RCPS) testbed. 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 widespread adoption of component-based software engineering has not only improved the maintainability of large complex code bases but also enabled structured analysis methods that rely on the principle of *compositionality* [1]: properties of a composed system can be derived from the properties of its components and connections. In prior work [2], we have considered one specific component model, DREMS (*Distributed Real-time Embedded Managed Systems*) [3], and developed a Colored Petri net-based (CPN) [4] 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 [5] 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 [6] to simulate and analyze resilient cyber-physical systems – consisting of 32 *Beaglebone Black* (BBB) development boards¹. We have chosen the lightweight *Robot Operating System* (ROS) [7] middleware layer and implemented a DREMS-style component model, ROS-MOD [8]. 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 *Resilient Cyber-Physical Systems* (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

¹BeagleBone Black Embedded Board: <http://beagleboard.org/BLACK/>

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 describes the operation execution semantics for our component model, presenting the need for our timing analysis methods. Section IV briefly summarizes our Colored Petri net-based analysis model in order to establish the level of refinement involved. Section V presents our experimental testbed, briefly describing the architecture and evaluation workflow. 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 well-established literature [9], [10], especially in the modeling and analysis of discrete event-driven systems. Models of such systems can be either *untimed* or *timed* models. Untimed models are those approximations where the order of the observed events are relevant to the design but the exact time instances when the state transitions occur are 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 Monotonic Scheduling analysis, there are very few usable tools that address the complex challenge of testing and certifying behaviors of complete, composed systems.

III. COMPONENT EXECUTION SEMANTICS

A *component 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 proportional-integral-derivative controller 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.

Figure 1 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 $t = t_{timer_cml}$. 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_{timer_cml} . Therefore, the request is waiting in the message

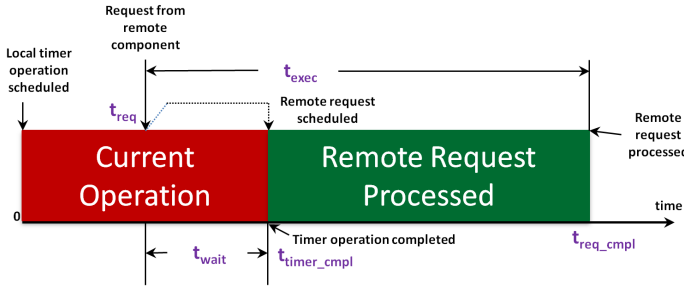


Fig. 1: Component Operation Execution Semantics: This figure shows the effects of the ROSMOD component scheduling on an incoming operation request. t_{req} represents the arrival time of a remote request. t_{wait} is the wait time of this request in the message queue while the current operation is still executing. t_{timer_cmpl} is the time stamp at which the current operation completes executing. At this time, the remote request is finally scheduled for execution. t_{req_cmpl} is the time stamp at which the remote request completes. The execution time, t_{exec} of this request is calculated as the difference in time stamps between t_{req_cmpl} and t_{req} .

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 executor threads of multiple components of equal priority, 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.

IV. CPN ANALYSIS MODEL

For the sake of brevity, a detailed description of both Colored Petri nets and our timing analysis model are not within the scope of this paper. The CPN analysis model consists of a collection of interacting sub-models, each responsible for modeling and simulating specific sub-systems in an application lifecycle e.g. thread scheduling, operation scheduling, thread execution, blocking and waiting times, timer expiries, and time progression. 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 [2] [5] e.g. processor utilization, execution time plots, deadline violations etc.

V. 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 [25] 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 implements the latter alternative. The RCPS testbed consists of 32 BBB embedded boards running Linux. The gigabit ethernet port of each BBB is connected to a programmable OpenFlow [26] enabled *Communication Network* switch. Each RCPS node is also connected to a *Physics Network* using a 10/100 USB-to-Ethernet adapter, since the BBBs only have one gigabit ethernet port. This network is connected to a *Physics Simulation Machine* running cyber-physical systems simulations. For further details on the architecture of this testbed, readers are referred to our earlier work [6].

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 ROSMOD, 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* (WCET), response times, processor utilization etc. are always conservative.

VI. EXPERIMENTAL EVALUATION

Experimental validation should demonstrate 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. 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 [5] 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 *remote method invocation* (RMI) application as shown in Figure 2. 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.

Note that in Figure 2, we only annotate isolated code blocks that take a fixed amount of execution time on a specific hardware architecture. These are the only measurements that we can reliably make with repeated testing and instrumentation. The client-side blocking delay is not measured because the number of factors responsible for this delay are numerous e.g. server's message queue state, scheduling non-determinism, network delays etc. In order to be able to predict this delay, we need to use state space analysis and search through the tree of possible executions to identify the worst-case blocking delay. This also means that our CPN model must capture and account for such delay-causing factors.

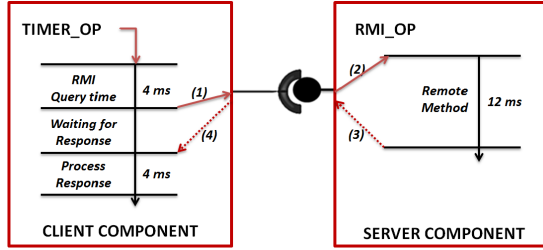


Fig. 2: RMI Application: A Periodically triggered client makes a remote method invocation to a server component. The assembly is annotated with estimated WCET on the different operational steps. The non-deterministic time here is the waiting time of the client, which is calculated using state space analysis.

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.

A. Time-triggered Operations

Time-triggered operations are an integral part of our component model. ROSMOD components are dormant by default. A timer has to trigger a inactive component for all subsequent interactions to take place. Since the ROSMOD component model supports various scheduling schemes on a single component message queue, this following test evaluates a priority first-in first-out 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 3 shows our experimental observations on a 5-timer example.

Since ROSMOD components are associated with a single executor thread and component operations are 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 4 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.

The analysis workflow, that has lead the results in Figure 4, is as follows: The time-triggered component is tested at real-time priority on the target platform, once for each timer which measures the pure execution time of each timer operation i.e. the time taken for each timer operation to execute on the target CPU without interruption. The WCET for each timer operation is then injected into our CPN analysis model and a composed timing analysis model is obtained. By performing bounded state space analysis, we derive the composed worst-case behavior of the component. Here, the execution times of each timer is affected by all other timers coupled with the execution semantics of the component model. These composed execution times will always be worse than the pure execution times due to factors like scheduling non-determinisms, context switching delays, and blocking behaviors. So, the results shown in Figure 4 are the composed timing analysis results that can now be compared against the experimental observations in Figure 3. As evident in these figures, the CPN analysis results are conservative estimates of the real execution.

B. Long-Running Operations

Our ROSMOD 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 ROSMOD component model does not support the *cancellation* of long-running 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 artificial intelligence-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 cooperating 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

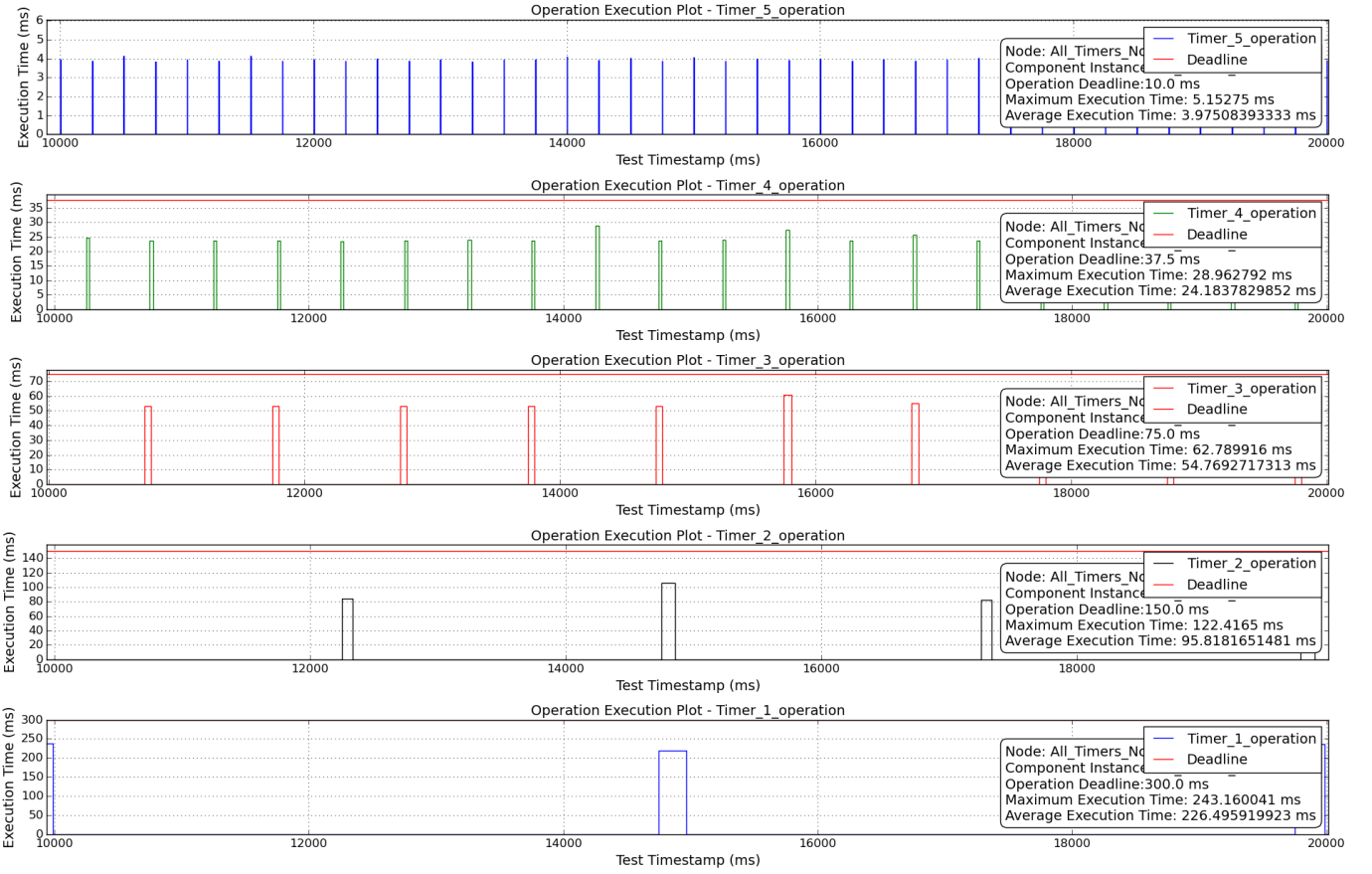


Fig. 3: Experimental Observation: Periodic Timers – Five periodic timers with different frequencies and different priorities executed by a single component.

bounded checkpoints, then the operation would execute one step at a time and suspend at such checkpoints if necessary. 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.

2) *Implementation and Results:* In each long-running operation, we, therefore, include a synchronous *checkpoint step*, as shown in Figure 5. 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 6 shows the *Software Model* for a component assembly with long running operations.

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 this operation is several times larger than the average execution time of all other operations. Figure 7 shows the execution time plot of this scenario, as measured on our testbed. Figure 8 shows our CPN analysis results for the same assembly. The plots represent the execution times of the various operation in the component assembly. As described in Section III, execution time refers to the duration of time taken for an operation to be marked as 'complete' once it is enqueued on the component message queue. This explains the intersections in the plots for *Name_Subscriber_operation* in *Component_1* as this is a subscriber port receiving messages on a same topic from two different publishers, in *Component_1* and *Component_2*. Secondly, it can be noted that the CPN predictions for *Timer_3_operation* show no variation between the worst-case execution time and the average-case execution time (381.0 ms). This is because *Component_3* is not collocated with any other component and executing alone. Thus, the operation is not affected by any scheduling non-determinism or context switching delays.

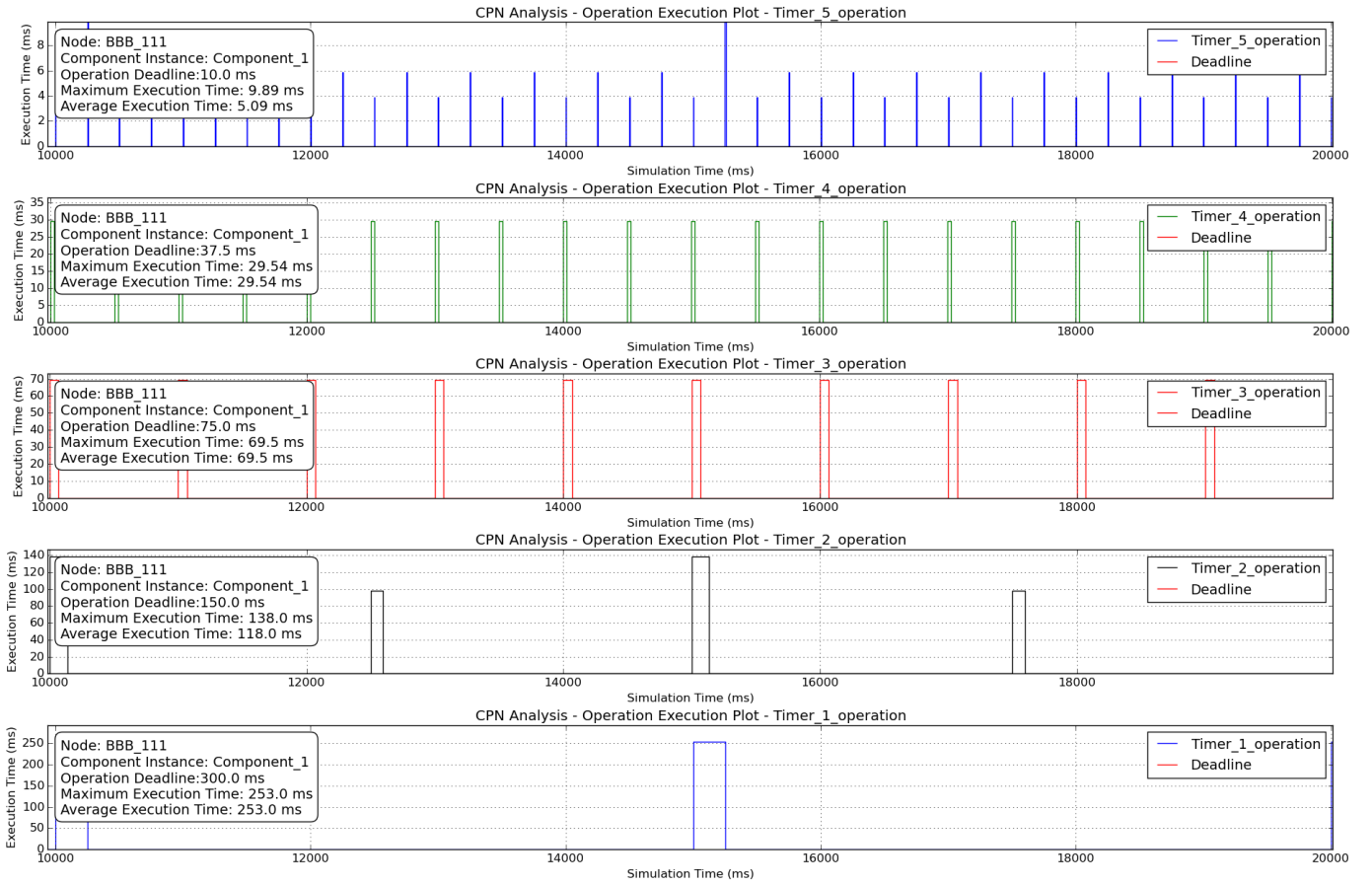


Fig. 4: CPN Analysis Results: Periodic Timers – CPN state space analysis and estimation of WCET for five periodically triggered times contained by a single component. Execution time measurements for each timer is made separately and composed in the CPN model. The resultant analysis presents a conservative estimation on the WCET of the composed system.

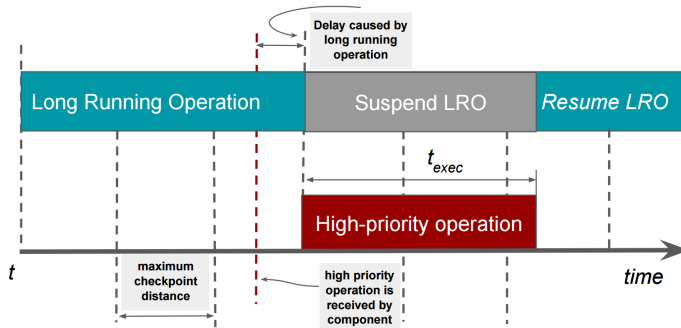


Fig. 5: Long Running Operations (LRO) Timing Diagram – The LRO is characterized by periodic checkpoints. At each checkpoint, the LRO checks to see if there are any higher priority operations waiting in the component message queue. If so, the LRO suspends till the higher priority operation completes and resumes and as soon as possible.

This exposes the conservative nature of the analysis. The Colored Petri net is an abstraction of the real execution behavior and makes several simplifications in order to obtain tractable analysis. Even still, there are analysis cases where the model is unable to perform as desired. For instance, the operating system (OS) scheduler uses a fixed-priority

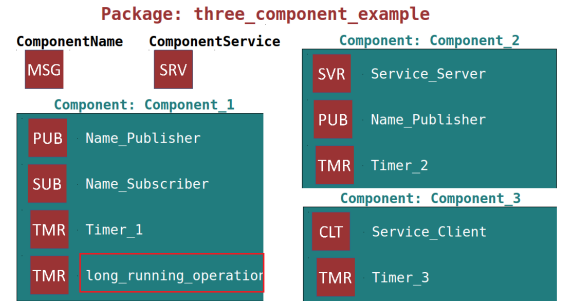


Fig. 6: Long Running Operation – The Software Model consists of three components. Component_1 contains a long running operation triggered by a sporadic timer.

preemptive scheme with Round-Robin conflict resolution i.e. when two equal-priority threads are ready to execute, one is chosen at random and subsequently, Round-Robin scheduling is enforced. In the analysis, when we increase the number of equal-priority component threads, the state space of the analysis will explode since there are $n!$ possible thread execution orderings for n equal priority threads executing on a device. The results of such an analysis may require searching a very

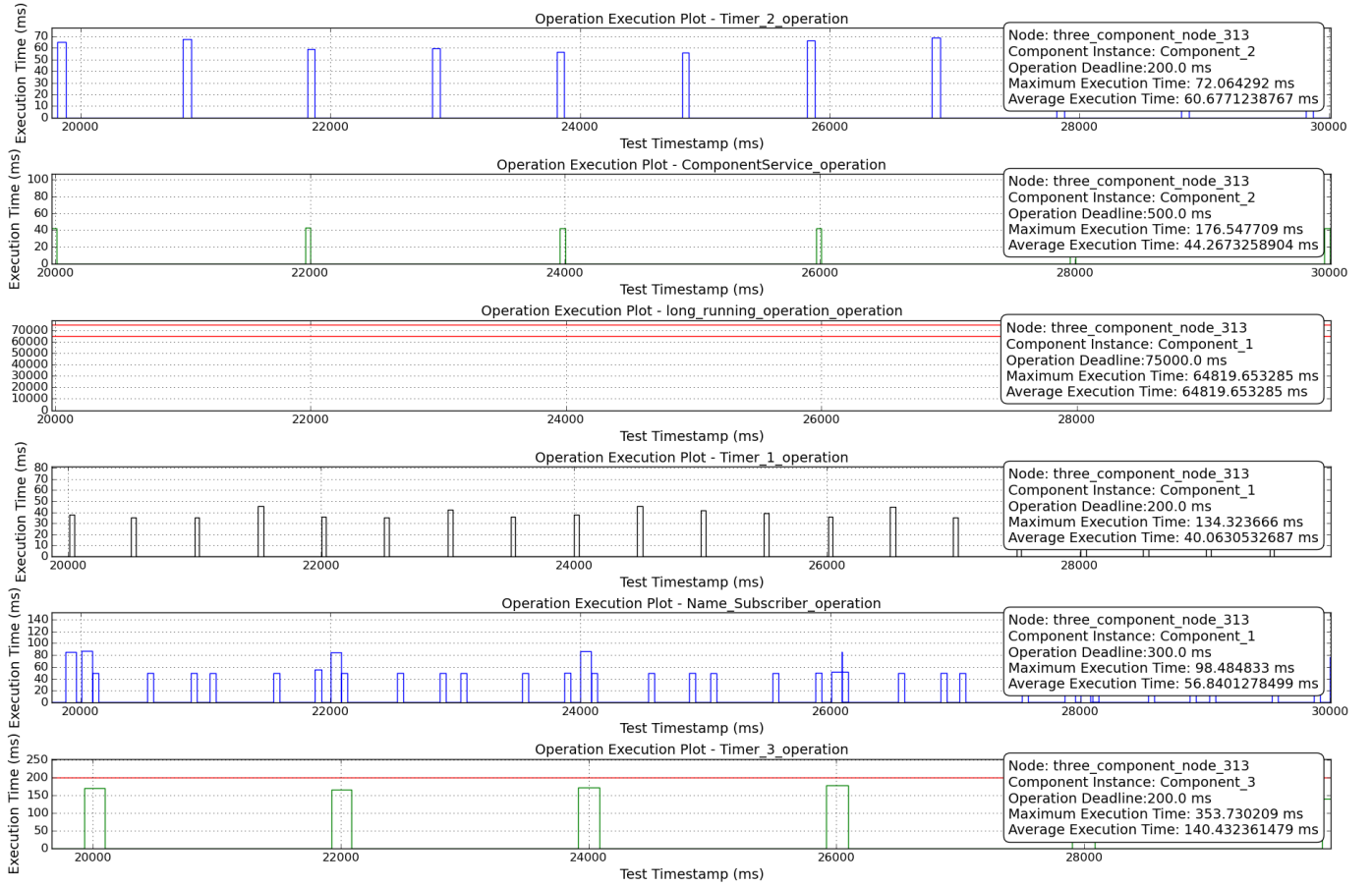


Fig. 7: Long Running Operations – Experimental Observation. The goal of this test is to ensure that the long-running operation can execute concurrently in other time-triggered operations in Component_1 and does not affect any timing properties. Periods and deadlines are chosen based on average-case performance.

large state space and may also cause a gross over-estimation of the execution times due to a single bad execution trace. This is a consequence of the simplified nature of the analysis, specifically the chosen OS scheduling model and the usage of worst-case execution times instead of other probabilistic methods. However, for strict safety-critical real-time systems that advocate predictable, deterministic behaviors, our analysis is still relevant.

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 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 Network Calculus-based analysis methods [27] 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] H. Jifeng, X. Li, and Z. Liu, “Component-based software engineering,” in *Theoretical Aspects of Computing—ICTAC 2005*. Springer, 2005, pp. 70–95.
- [2] 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.
- [3] 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.
- [4] K. Jensen and L. M. Kristensen, *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

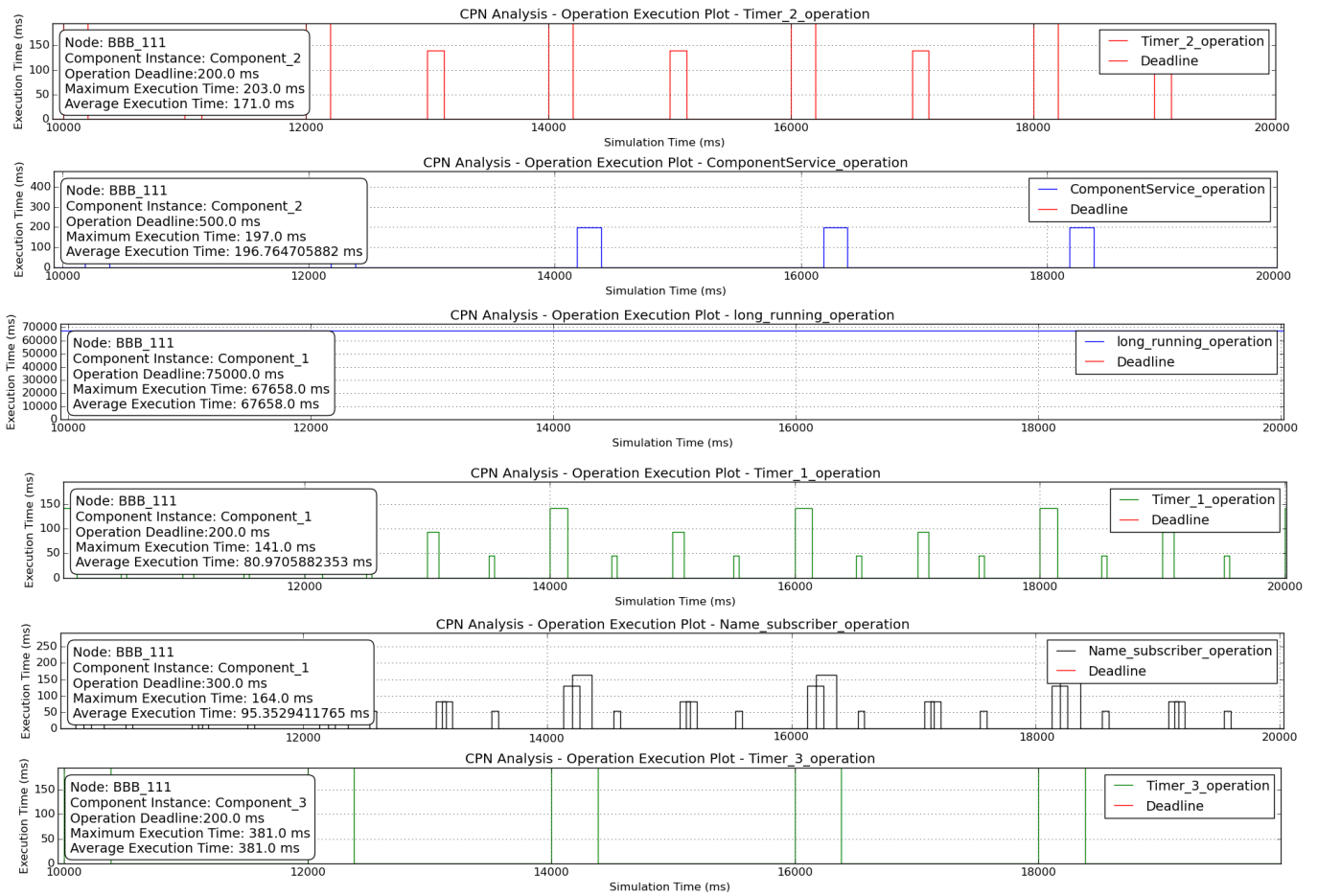


Fig. 8: Long Running Operations – CPN Analysis Estimates. The CPN receives pure execution times of all operations in the component assembly from which a composed analysis is performed. The WCET plot is the representative worst-case execution trace obtained from state space analysis.

- [5] 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.
- [6] 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.
- [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] "Simulink," <http://www.mathworks.com/products/simulink/>.
- [26] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.
- [27] 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>