Integrated Analysis of Temporal Behavior of 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*—Integrated analysis of temporal behavior for distributed real-time embedded (DRE) systems is an important design-time step needed to verify safe and predictable system operation at run-time. In earlier work, we have shown a Colored Petri Net-based (CPN) approach to modeling and analyzing component-based DRE systems. In this paper, we present new CPN-based modeling approaches and advanced state space methods that improve on the scalability and efficiency of the analysis. The generality of the modeling principles used show the applicability of this approach to a wide range of systems.**

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

I. Introduction

Real-time systems, by definition, must meet operational deadlines. These deadlines constrain the amount of time permitted to elapse between a stimulus provided to the system and a response generated by the system. Delayed responses or missed task deadlines can have catastrophic effects on the function of the system, especially in the case of safety- and mission-critical applications. This is the primary motivation for design-time schedulability analysis and verification of systems.

There is a wealth of existing literature studying real-time task scheduling theory and timing analysis in uniprocessor and multiprocessor systems [1], [2]. There are also several modeling, schedulability analysis and simulation tools [3], [4], [5], [6] that address heterogeneous challenges in verifying real-time requirements although many such tools are appropriate only for certain task models, interaction patterns, scheduling schemes, or analysis requirements. For component-based architectures, model-based system designs are usually expressed in formal domain such as timed automata [7], [8], controller automata [9], high-level Petri nets [10] etc. so that existing analysis tools such as UPPAAL [11] or CPN Tools [12] can be used to verify either the entire system or its compositional parts. But, it is also evident that many of the existing schedulability analysis tools, though grounded in theory are not directly applicable to all system designs, especially with respect to domain-specific properties such as component interaction patterns, distributed deployment, time-varying communication networks etc.

To be useful, the analysis tools need to be tightly integrated with the target domain: the concurrency model used by the system. The classic thread-based concurrency model (with generic synchronization primitives) is too low-level and too generic, it is hard to use, and hard to analyze. For pragmatic reasons, more restrictive, yet useful concurrency models are needed for which dedicated analysis tools can be developed. Our previous efforts [13] were directed at this challenge. The target domain for that study was the DREMS component model [14] which is the foundation of a software infrastructure addressing challenges in the design, development and deployment of component-based flight software for fractionated spacecraft. The physical nature of such systems require strict, accurate and pessimistic timing analysis at design-time to avoid catastrophic situations. DREMS is implemented as a design-time tool suite and a run-time software platform that is enhanced by a component (concurrency) model with well-defined execution semantics. The platform relies on a temporally partitioned task scheduling scheme, a non-preemptive component-level operations scheduler, support for various communication and interaction patterns; all deployed on a distributed hardware platform.

Our contributions in this paper target efficient modeling and analysis techniques of temporal behavior for component-based applications that form distributed real-time embedded systems, such as DREMS.

1) We present an approach for modeling the 'business logic': the operational behavior of each component in an application. The model uses a sequence of timed steps that are executed by a component operation, including steps the include interactions with other components. This approach enables abstracting the details of the middleware, while representing the temporal behavior of the component business logic.
2) We also present improvements to the CPN-based modeling approach that enables better analysis performance and scalability. These rely on heuristics that manage time variables and state space data structures more efficiently.
3) We also present advanced state space analysis methods

and tools applied on the modeled system to reduce analysis time on medium to large-scale systems.

The rest of this paper is organized as follows. Section II presents related research, reviewing and comparing existing analysis tools and formal methods. Sections III briefly describes the DREMS architecture, specifically the concepts of interest that are covered by the timing analysis tool. Section IV describes the business logic modeling approach to capture the operational behavior of components in the application. Section V describes the analysis improvements we were able to achieve with structural changes to the analysis model. This section also briefly describes the application of advanced state space analysis methods that enable efficient state space searches while reducing the state space size and overall memory consumption. Section VI evaluates possible extensions to this work before concluding with Section VII.

II. Related Research

Verification of component-based systems require significant information about the application assembly, interaction semantics, and real-time properties. This information is primarily derived from the design model although many real-time metrics are not explicitly modeled. Using model descriptors, [15] describes interaction semantics and real-time properties of components. Using the MAST modeling and analysis framework [3], [16], schedulability analysis and priority assignment automation is supported. Event-driven models are separated into several *views* which are similar to hierarchical pages in CPN. Analysis efforts include the calculation of response times, blocking times and slack times.

High-level Petri nets are a powerful modeling formalism for concurrent systems and have been integrated into many modeling tool suites for design-time verification. General-purpose AADL models have been translated into Symmetric nets for qualitative analysis [17] and Timed Petri nets [18] to check real-time properties such as deadline misses, buffer overflows etc. Similar to [18], our CPN-based analysis also uses bounded observer places [19] that observe the system behavior for property violations and prompt completion of operations. However, [18] only considers periodic threads in systems that are not preemptive. Our analysis is aimed at a combination of preemptive and non-preemptive hierarchical scheduling with higher-level component interaction concepts. separately.

Several analysis approaches present tool-aided methodologies that exploit the capabilities of existing analysis and verification techniques. In the verification of timing requirements for composed systems, [20] uses the OMG UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) modeling standard and converts high-level design into MAST output models for concrete schedulability analysis. In a similar effort, AADL models are translated into real-time process algebra [21] reducing schedulability analysis into a deadlock detection problem searching through state spaces and providing failure scenarios as counterexamples. Symbolic schedulability analysis has been performed by translating the task sets into a network timed automata, describing task arrival patterns and various scheduling policies. TIMES [5] calculates worst-case response times and scheduling policies by verifying timed automata with UPPAAL [11] model checking.

In order to analyze hierarchical component-based systems, the real-time resource requirements of higher-level components need to be abstracted into a form that enables scalable schedulability analysis. The authors in [22] present an algorithm where component interfaces abstract the minimum resource requirements of the underlying components, in the form of periodic resource models. Using a single composed interface for the entire system, the component at the higher level selects a value for operational period that minimizes the resource demands of the system. Such refinement is geared towards minimum waste of system resources.

III. Background

The target architecture for timing analysis is *DREMS* [23], [14]. DREMS was designed and developed for a class of distributed real-time embedded systems that are remotely managed and have strict timing requirements. DREMS is a software infrastructure for the design, implementation, deployment, and management of component-based real-time embedded systems. The infrastructure includes design-time modeling tools [24] that integrate with a well-defined and fully implemented component model [25], [13] used to build component-based applications. Rapid prototyping and code generation features coupled with a modular run-time platform automate tedious aspects of software development and enable robust deployment and operation of mixed-criticality distributed applications. The formal modeling and analysis method presented in this paper focuses on applications built using this foundational architecture.

A. Component Operations Scheduler

DREMS applications are built by assembling and composing re-useable units of functionality called *Components*. Each component is characterized by a (1) set of communication ports, a (2) set of interfaces (accessed through ports), a (3) message queue, (4) timers and state variables. Components interact via publish/subscribe and synchronous or asynchronous remote method invocation (RMI and AMI) services provided by the middleware. Each component interface exposes one or more *operations* that can be invoked due to the arrival of a message or a method invocation or the expiration of a timer. Every operation request coming from an external entity reaches the component through its message queue. This is a priority queue maintained by a component-level scheduler that schedules operations for execution. When ready, a single *component executor thread* per component will be released to execute the operation requested by the front of the component's message queue. The operation runs to completion, hence component execution is always single-threaded. Note however that the multiple components *can* be executed concurrently.

B. Operating System Scheduler

DREMS components are grouped into processes that may be assigned to ARINC-653 [26] styled temporal partitions, implemented by the DREMS operating system scheduler. Temporal partitions are periodic, fixed intervals of the CPU's time. Threads associated with a partition are scheduled only when the partition is active. This enforces a temporal isolation between threads assigned to different partitions and assigns a guaranteed slice of the CPU's time to that partition. The repeating partition windows are called *minor frames*. The aggregate of minor frames is called a *major frame*. The duration of each major frame is called the *hyperperiod*, which is typically the lowest common multiple of the partition periods. Each minor frame is characterized by a period and a duration. The duration of a partition defines the amount of time available per hyperperiod to schedule all threads assigned to that partition. Each *node* in a network runs an OS scheduler, and the temporal partitions of the nodes are assumed to be synchronized, i.e. all hyperperiods start at the same time.

C. Colored Petri Nets

Petri Nets [27] 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, M_0) 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 M_0 is the initial marking of the net. Places hold a discrete number of markings called tokens. Tokens often represent resources in the modeled system. A transition can legally fire when all of its input places have necessary number of tokens.

With Colored Petri Nets (CPN) [28], tokens contain values of specific data types called *colors*. Transitions in CPN are enabled for firing only when valid colored tokens are present in all of the typed input places, and valid arc bindings are realized to produce the necessary colored tokens on output places. The firing of transitions in CPN can check for and modify the data values of these colored tokens. Furthermore, large and complex models can be constructed by composing smaller sub-models as CPN allows for hierarchical description.

1) The Choice of Colored Petri Nets: One of the primary reasons for choosing Colored Petri Nets over other high-level Petri Nets such as Timed Petri Nets or other modeling paradigms like timed automata is because of the powerful modeling concept made available by token colors. Each colored token can be a heterogeneous data structure such as a *record* that can contain an arbitrary number of fields. This enables modeling within a single *color-set* (C-style struct) system properties such as temporal partitioning, component interaction patterns, and even distributed deployment. The token colors can be inspected, modified, and manipulated by the occurring transitions and the arc bindings. Component properties such as thread priority, port connections and real-time requirements can be easily encoded into a single colored token, making the model considerably concise. In Section V, we will discuss in more detail how some of the modeling concepts/changes we have made affect and improve the efficiency of analysis.

IV. Modeling Temporal Behavior

The execution of component operations service the various periodic or aperiodic interaction requests coming from either a timer or other connected (possibly distributed) components. Each operation is written by an application developer as a sequence of execution *steps*. Each step could execute a unique set of activities, e.g. perform a local calculation or a library call, initiate an interaction with another component, process a response from external entities, and it can have data-dependent, possibly looping control flow, etc. The behavior derived by the combination of these steps contribute to the worst-case execution of the component operation. The behavior may include non-deterministic delays due to component interactions while being constrained by the temporally partitioned scheduling scheme and hardware resources. This section briefly describes the various aspects of this behavior specification that are general enough to be applicable to a range of component-based systems.

```
(* Business Logic syntax in Extended Backus-Naur Form *)
business_logic          =   'Do', ws, operation_name, ws
                            '[', operation_priority, operation_deadline, ']', '{', { functional_step }, '};' ;
operation_name          =   ID ;
operation_priority      =   INT ;
operation_deadline      =   INT ;
functional_step         =   {sequential_code_block | rmi_call | ami_call | dds_publish | dds_pull_subscribe |
                                                    dds_push_subscribe | loop} ;
sequential_code_block   =   INT, ';' ;
rmi call                =   'RMI', ws, receptacle_port, '.', remote_operation, '[' query_time, processing_time ];' ;
ami call                =   'AMI', ws, receptacle_port, '.', remote_operation, '[' query_time, processing_time ];' ;
dds publish             =   'DDS_Publish', ws, dds_port, '.', topic, '[', publish_time, ];' ;
dds pull subscribe      =   'DDS_Pull_Subscribe', ws, dds_port, '.', topic, '[', processing_time, ];' ;
dds_push_subscribe      =   'DDS_Push_Subscribe', ws, dds_port, '.', topic, '[' processing_time, ];' ;
loop                    =   'LOOP', ws, '[', count, ']', ws, '{', {functional_step}, '};' ;
receptacle_port         =   ID ;
remote_operation        =   ID ;
dds_port                =   ID ;
topic                   =   ID;
query_time              =   INT ;
processing_time         =   INT ;
publish_time            =   INT ;
count                   =   INT;
```

Fig. 1: Modeling the Business Logic of Component Operations

Figure 1 shows the Extended Backus-Naur form representation of the grammar used for modeling the business logic of component operations. The symbol *ID* represents identifiers, a unique grouping of alphanumeric characters/terminal symbols and the symbol *INT* represents positive integer digits. Each operation is characterized by a unique name, a priority and a deadline. The priority is an integer used to resolve scheduling conflicts between operations *provided* by the same component when requests from external entities are received. The arbitration is handled by the component-level scheduler. The deadline of the operation is the worst-case amount of time that can elapse after the operation is marked as *ready*. The business logic of every component operation is modeled as a sequence of functional steps, each with an assigned worst-case execution time. We broadly classify these steps into (1) blocks of sequential code, (2) peer-to-peer synchronous and asynchronous remote calls, (3) anonymous publish/subscribe

distribution service calls, (4) blocking and non-blocking I/O interactions and (5) bounded control loops.

Notice the integration of timing properties such as worst-case function call times (*query_time*), worst-case argument processing times (*processing_time*) and DDS publish times (*publish_time*). If these expected delays are set to zero, the analysis will execute these interactions in a single synchronous step taking no time. However, in reality these steps still take a non-zero amount of time to execute. Therefore, if such metrics are not known then these values can be set to zero and an overall worst-case execution time can be set per operation. This is the maximum amount of time that can elapse after the component operation has begun to execute. This time will include all component interactions and network delays that affect the operation's execution.

V. Analysis Techniques

A. Handling Time

The CPN-based analysis consists of executing a simulation of the model and constructing a state space data structure for the system (for a finite horizon), and then performing queries on this data structure. This is automated by CPN Tools. The first improvement over the basic CPN approach is in how we handle time. Although it is true that CPN and similar extensions to Petri Nets such as Timed Petri Nets inherently have modeling concepts for simulation time, we explicitly model time as an integer-valued *clock* color token in CPN. There are several reasons for this choice.

Firstly, this is an extension to our previous arguments about choosing Colored Petri Nets. Modeling the OS scheduler clock as a colored token allows for extensions to its data structure such as (1) intermediate time stamps and internal state variables, and (2) adding temporal partitioning schemes like the (time-partitioned) ARINC-653 [26] scheduling model (Figure 2). These extended data structure fields can be more easily manipulated and used by the model transitions during state changes, allowing for richer modeling concepts that would not be easily attainable using token representations provided by Timed Petri Nets. The ability to pack colored tokens with rich data structures also reduces the total number of colors required by the complete model. This quantitative measure directly influences the reduced size of the resultant state space. The downside of this approach to modeling is that we have to choose a time quantum. But in practical systems this is usually not a problem, as the low-level scheduling decisions are taken by an OS scheduler based on a time scale with a finite resolution. We have chosen 1 msec as the quantum (corresponding to the typical 1KHz scheduler in Linux), but it can be easily changed.

```
1`[{clock_node="Sat1", clock_value=0, schedule=[{part_name="Part1", exec_t=0, dur=20, pr=40, off=0},
                                                {part_name="Part2", exec_t=0, dur=20, pr=40, off=20},
                                                {part_name="Part3", exec_t=0, dur=20, pr=40, off=40}]}]
```

Fig. 2: A Clock Token with Temporal Partitioning

Secondly, modeling time as a token allows for smarter time progression schemes that can be applied to control the pace of simulation. If we did not have such control over time, the number of states recorded for this color token would eventually explode and itself contribute to a large state space. In order to manage this complexity, we have devised some appropriate *time jumps* in specific simulation scenarios.

If the rate at which time progresses does not change, then for a 1 msec time resolution, S seconds of activity will generate a state space of size: $SS_{size} = \sum_{i=1}^{S*1000} TF_{t_i}$ where TF_{t_i} is the number of state-changing CPN transition firings between t_i and t_{i+1}. This large state space includes intervals of time where there is no thread activity to analyze either due to lack of operation requests, lack of ready threads for scheduling, or due to temporal partitioning. During such idle periods, it is prudent to allow the analysis engine to *fast-forward* time either to (1) the next node-specific clock tick, (2) the next global timer expiry event, or (3) the next activation of the node-specific temporal partition (whichever is earliest and most relevant). This ensures that the generated state space tree is devoid of nodes where there is no thread activity.

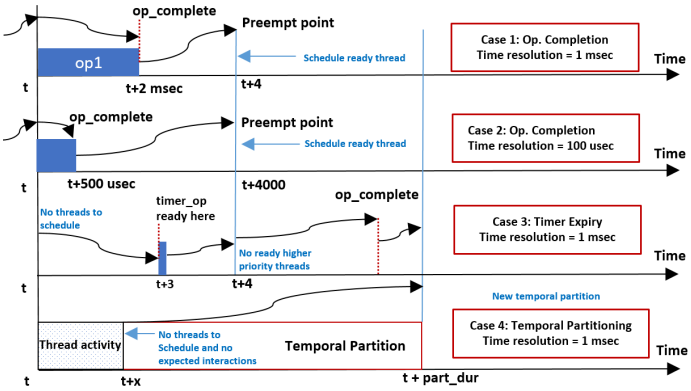

Fig. 3: Dynamic Time Progression

Figure 3 illustrates these time jumps using 4 scenarios. Assuming the scheduler clock ticks every 4 msec, Case 1 shows how time progression is handled when an operation completes 2 msec into its thread execution. At time t, the model identifies the duration of time left for an operation to complete. If this duration is earlier than the next preempt point, then there is no need to progress time in 1 msec increments as no thread can preempt this currently running thread till time t + 4 msec. Therefore, the *clock_value* in Figure 2 progresses to time t + 2 msec, where the model handles the implications of the completed operation. This includes possibly new interactions and operation requests triggered in other components. Then, time is forced to progress to the next preempt point where a new candidate thread is scheduled. This same scenario is illustrated in Case 2 when the time resolution is increased to 100 usec instead of 1 msec. Notice that the number of steps taken to reach the preempt point are the same, showing how the state space doesn't have to explode simply because the time resolution is increased. Case 3 illustrates the scenario where at time t, the scheduler has no ready threads

to schedule since there are no pending operation requests but at time t + 3 msec, a component timer expires, triggering an operation into execution. Since timers are maintained in a global list, each time the *Progress_Time* transition checks its firing conditions, it checks all possible timers that can expiry before the next preempt point. So, at time t when no threads are scheduled, the model immediately jumps to time t + 3. This scenario also shows that if the triggered operation does not complete before the preempt point *and* there are no other ready threads or timer expiries that can be scheduled, the clock value jumps to the operation completion. It must be noted here that this case is valid only because the DREMS architecture we have considered uses a non-preemptive operation scheduling scheme. Lastly, Case 4 shows time jumps working with temporal partitioning. At some time t + x, the model realizes the absence of ready threads and does not foresee any interaction requests from other components, then it safely jumps to the end of the partition without stepping forward in 1 msec increments. This time progression directly shows how the state space of the system execution reduces while still preserving the expected execution order, justifying our choice of modeling time as a colored token using CPN.

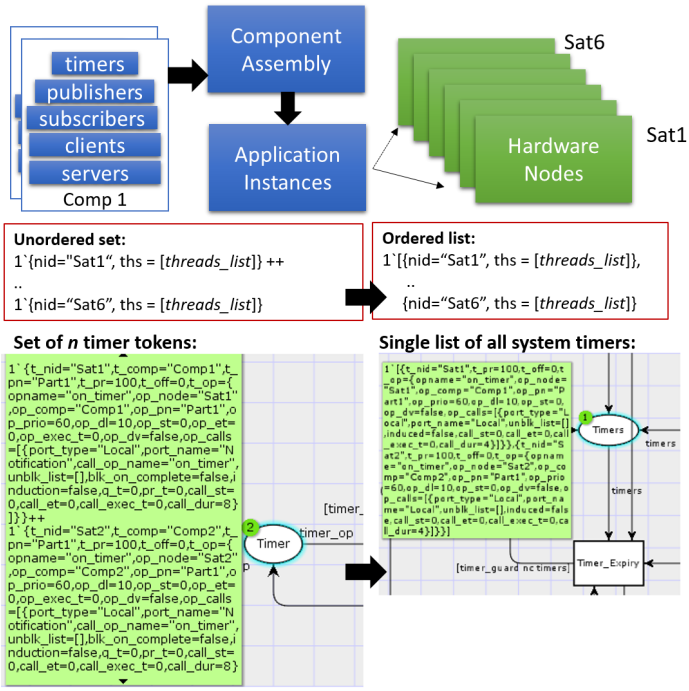

Fig. 4: Structural Reductions in CPN

B. Distributed Deployment

The second structural change to the analysis model is in how distributed deployments are modeled and simulated. Early designs on modeling and analysis of distributed application deployments [13] included a unique token per CPN place for each hardware node in the scenario. Since the individual *node* tokens are independent and unordered, there is a nondeterminism in the transition bindings when choosing a hardware node to schedule threads in. For instance, if there are 2 hardware nodes in the deployment with ready threads on both nodes, then either node can be chosen first for scheduling threads leading to two possible variations of the model execution trace. Therefore the generated state space would exponentially grow for each new hardware node. In order to reduce this state space and improve the search efficiency, we have merged hardware node tokens into a single *list* of tokens instead of a unassociated grouping of individual node tokens. This approach is inspired by the symmetry method for state space reduction [29].

Figure 4 illustrates this structural reduction. Consider a distributed deployment scenario with an instance of a DREMS application deployed on each hardware node, Sat1 through Sat6. Components *Comp1* and *Comp2* are triggered by timers, eventually leading to the execution of component operations (modeled as shown in Figure 1). If all the timer tokens in the system were modeled individually, the transition *Timer_Expiry* would non-deterministically choose one of the two timer tokens that are ready to expire at *t=0*. However, if the timers are maintained as a single list, then this transition (1) consumes the entire list, (2) identifies all timers that are ready to expire, (3) evaluates the timer expiration function on all ready timers, (4) propagates the output *operation* tokens to the relevant component message queues in a single firing. This greatly reduces the tree of possible transition firings and therefore the resultant state space. Also, if there is no non-determinism in the entire system, i.e., there is a distinct ordering of thread execution, then this model can be scaled up with instantiating the application on new hardware nodes with no increase in state space size. This is because all of the relevant tokens on all nodes are maintained as a single list that is completely handled by a single transition firing.

An important implication to the above structural reduction is that the simulation of the entire system now progresses in synchronous steps. This means that at time 0, all the timers in all hardware nodes that are ready to expire will expire in a single step. Following this, all operations in all component message queues of all these nodes are evaluated together and appropriate component executor threads are scheduled together. When these threads execute, time progresses as described in Section V-A, moving forward by the minimum amount of time that can be fast-forwarded.

C. Advanced State Space Analysis Methods

State space analysis techniques have been successfully applied with Colored Petri Nets in a variety of practical scenarios and industrial use cases [30], [31]. The basic idea here is to compute all reachable states of the modeled concurrent system and derive a directed graph called the *state space*. The graph represents the tree of possible executions that the system can take from an initial state. It is possible from this directed graph to verify behavioral properties such as queue overflows, deadline violations, system-wide deadlocks and even derive counter examples when arriving at inconsistent states.

The variety of CPN-specific state space reduction techniques [32], [33] developed in recent times has significantly broadened the class of systems that can be verified. In order to easily apply such techniques to our analysis model, we use the ASAP [34] analysis tool. The tool provides for several search algorithms and state space reduction techniques such as the *sweep-line method* [35] which deletes already visited state space nodes from memory, forcing on-the-fly verification of temporal properties. The main advantage of such a technique is the amount of memory required by the analysis to verify useful properties for large models.

The sweep line method for state space reduction is used to check for important safety properties such as lack of deadlocks, timing violations etc. using user-defined model-specific queries. Practical results enumerated in [35] show improvements in time and memory requirements for generating and verifying bounded state spaces. The method relies of discarding generated states on-the-fly by performing verification checks during state space generation time. Any state that does not violate system properties can be safely deleted. Another advantage of this method of similar reduction methods such as bit-state hashing [36] is that a complete state space search is guaranteed.

In order to illustrate the utility of such state space reduction techniques, we consider a large-scale deployment. Figure 5 shows the generated CPN model for a domain-specific DREMS application. This is a scaled-up variant of several satellite cluster examples we have used in previous publications [14], [13]. The example consists of a group of communicating satellites hosting DREMS applications. The component assembly for this application consists of 100 interacting components distributed across 10 computing nodes, many of which are triggered by infrastructural timers. Notice in Figure 5 how there is only one token in each of the main CPN places, as described in Section V-B. All of the component timers are appended to the list maintained in *Timers* place. Similarly, all node-specific clock tokens are maintained in place *Clocks*.

At time *t=0*, before the simulation is kicked off, the transition *Establish_Order* generates the non-deterministic set of thread execution orders that are possible given the configuration of the clock token. This may be a potentially large set depending on the number of threads of equal priority in each partition. Once this tree of possible orders is established, the complete set of timers that are ready to expire are evaluated. Each timer expiry manifests as an operation request and each callback operation modeled using the grammar shown in Figure 1. Once the operations are ready to execute, the highest priority component thread with a pending operation request is chosen for execution. This thread scheduling happens on all hardware nodes. When each thread executes, new interactions may occur as a consequence of the execution. For instance, if a component thread executes a timer operation in which the component publishes on a global topic, the consequence of this action would include a set of callback operation requests on all components that contain subscribers to that global topic. Lastly, all running threads are evaluated to identify the minimum amount of time that can be safely fast-forwarded in each node. If the running component threads are independent or symmetrical, then the maximum possible time progression is up to the end of the temporal partition. Note here that temporal partition in the deployment can be set to an empty list which simply removes the partitioning constraint and treats all component threads on a node as candidate threads for execution. The above sequence of transitions repeat for as long as there is a timer expiry, a pending operation request or an unfinished component interaction.

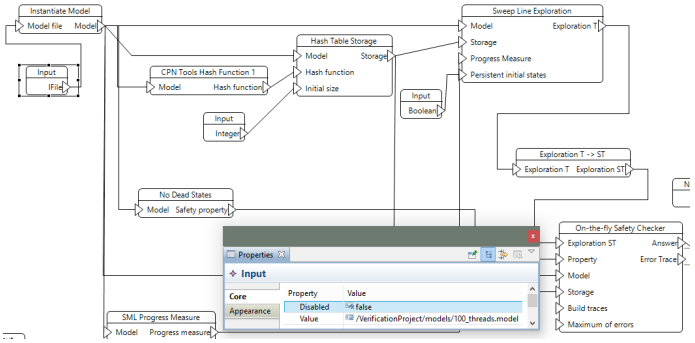

Fig. 6: Sweep-Line Method

Using the CPN Tools in-built state space analysis tool, a bounded state space was generated reaching up-to 20 hyper-periods of component thread activity. This bounded generation took 36 minutes on a typical laptop. Our goal with such an example is to evaluate the effectiveness and utility of state space reduction techniques with respect to speed and memory usage. Figure 6 shows a simple block diagram of the sweep-line method as configured in ASAP. Performing on-the-fly verification checks for lack of dead states in the analysis model, results indicate lack of system-wide deadlocks due to blocking behaviors triggered by RMI-style synchronous peer-to-peer interaction patterns. Figure 7 shows analysis results obtained from a *Verification Job* executed in the tool. Notice the on-the-fly verification taking less than 10 minutes to perform deadlock checks on this sample deployment. Using the *Palette* in ASAP, several standard ML (SML) user queries can be created to check for domain-specific properties.

It must be noted here that this improved result is not only because of the efficient state space search but also because of symmetry-based structural reduction discussed in Section V-B. If not for this reduction, the state space search requirements would exponentially grow for each new hardware node added to the deployment.

D. Discussion

1) Conservative Results: Using estimates of worst-case execution time for component operations is motivated by the need to make exaggerated assumptions about the system behavior. Pessimistic estimates are a necessary requirement when verifying safety-critical DRE systems. Schedulability analysis with such assumptions should provide strictly conservative

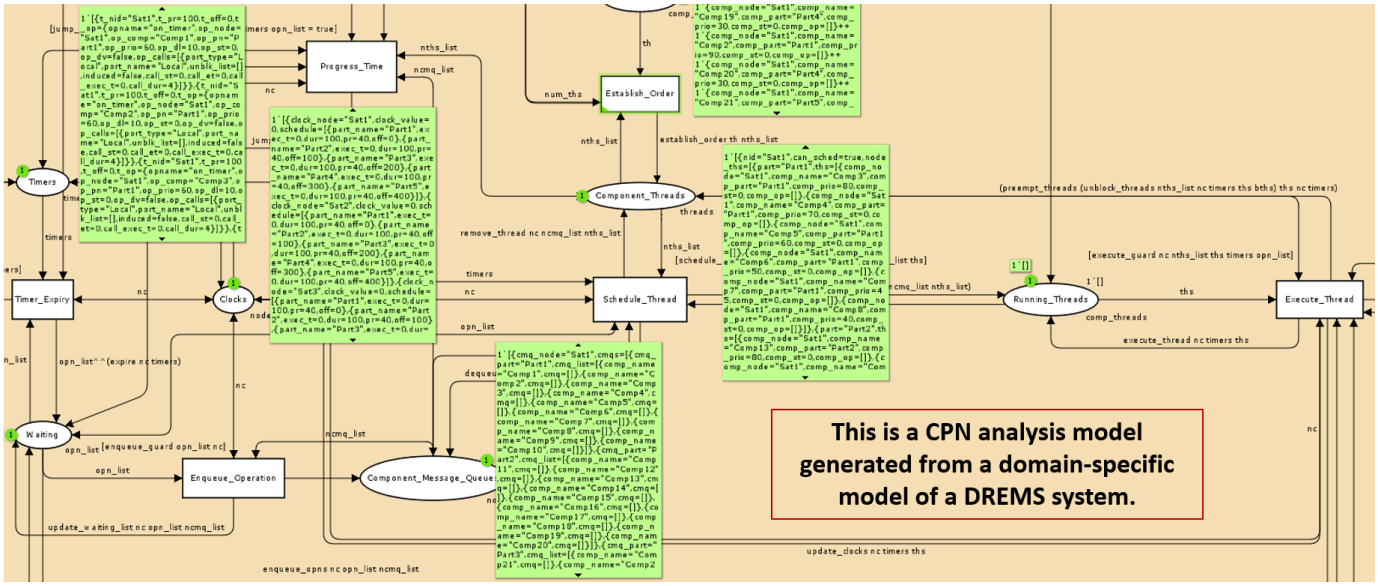

Fig. 5: Generated CPN model for a Distributed Application Deployment

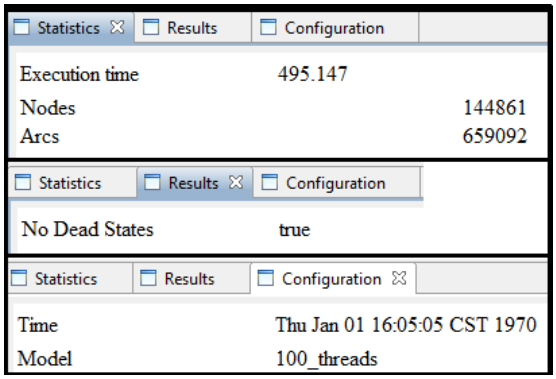

Fig. 7: Dead States Checking in a Component-based application

results. This means that (1) if the analysis results show the possibility of a deadline violation but the deployed system does not, the obtained result is a conservative one as it assumes worst-case behavior, and (2) if the analysis results do not show any timing violations but the deployed system violates response time requirements, deadlines etc., then the analysis does not provide a conservative result and has failed to verify system behavior.

In order to guarantee conservative results, the analysis must include worst-case behaviors of all system-level threads that run at higher priority than component threads and are not necessarily modeled by the design-time tools. In our analysis, these threads are grouped into a set of critical processes with approximations made to simulate the behavior of system-level threads such as (1) globally periodic CPU utilization, (2) CPU utilization for some WCET per partition. The best approximation is chosen based on the expected behavior of such critical processes. Best-effort processes are ignored as they always run at a priority lower than the lowest-priority component thread.

VI. Future Work

The DREMS component communication is facilitated by a time-varying network. The bandwidth provided by the system therefore predictably fluctuates between a minimum and a maximum during the orbital period of the satellites. Currently, our analysis associates each operational step with a fixed worst-case network delay. We are working on introducing places in the CPN model that capture the *network profile* of a deployment so that the communication delays during queries vary with time leading to tighter bounds on predicted response times.

Secondly, we are also investigating the utility of this approach for fault-tolerant and self-adaptive systems. Integrating this analysis with a run-time resilience engine, configuration changes determined by a fault mitigating solver can be subsequently checked for timing anomalies before settling on a specific reconfiguration strategy.

VII. Conclusions

Mobile, distributed real-time systems operating in dynamic networking environments, and running mission-critical applications must still satisfy strict timing requirements to operate safely. To reduce the development and integration complexity for such systems, component-based design models are being increasingly used. Appropriate analysis models are required to study the structural and behavioral complexity in such designs.

This paper presented new Colored Petri net-based modeling approaches to capture the architecture and temporal behavior of such component-based applications for both qualitative and quantitative schedulability analysis. The key idea behind the analysis is the precise modeling of the component execution semantics in the analysis tool. The analysis method is modular, extensible and intuitive and there are sufficient support tools to enable state space analysis and verification for medium to large

size design models. We investigated the utility of structural reduction and advanced state space analysis techniques in order to realize a more efficient and scalable analysis.

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] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings, "Fixed priority pre-emptive scheduling: An historical perspective," *Real-Time Systems*, vol. 8, no. 2-3, pp. 173–198, 1995. [Online]. Available: http://dx.doi.org/10.1007/BF01094342

[2] L. Sha, T. Abdelzaher, K.-E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, "Real time scheduling theory: A historical perspective," *Real-Time Syst.*, vol. 28, no. 2-3, pp. 101–155, Nov. 2004. [Online]. Available: http://dx.doi.org/10.1023/B:TIME.0000045315.61234.1e

[3] M. G. Harbour, J. J. G. Garcia, J. C. P. Gutierrez, and J. M. D. Moyano, "Mast: Modeling and analysis suite for real time applications," in *In 13th Euromicro Conference on Real-Time Systems*, 2001, p. 125.

[4] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: A flexible real time scheduling framework," in *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-time &Amp; Distributed Systems Using Ada and Related Technologies*, ser. SIGAda '04. New York, NY, USA: ACM, 2004, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/1032297.1032298

[5] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Times: a tool for schedulability analysis and code generation of real-time systems," in *Formal Modeling and Analysis of Timed Systems*. Springer, 2004, pp. 60–72.

[6] P. Derler, T. H. Feng, E. A. Lee, S. Matic, H. D. Patel, Y. Zhao, and J. Zou, "Ptides: A programming model for distributed real-time embedded systems," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-72, May 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-72.html

[7] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[8] G. Macariu and V. Cretu, "Timed automata model for component-based real-time systems," in *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*, March 2010, pp. 121–130.

[9] Y. Zhang, H. Lin, and G. Li, "Emerald: An automated modeling and verification tool for component-based real-time systems," in *Quality Software (QSIC), 2012 12th International Conference on*, Aug 2012, pp. 120–123.

[10] A. Masri, T. Bourdeaud_huy, and A. Toguyeni, "A component-based approach based on High-Level Petri Nets for modeling Distributed Control Systems," *International Journal On Advances in Intelligent Systems*, vol. 2, no. 2 et 3, pp. 335–353, Sep. 2009. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00801500

[11] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems," in *Proc. of Workshop on Verification and Control of Hybrid Systems III*, ser. Lecture Notes in Computer Science, no. 1066. Springer–Verlag, Oct. 1995, pp. 232–243.

[12] 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

[13] *Colored Petri Net-based Modeling and Formal Analysis of Component-based Applications*, 2014. [Online]. Available: http://ceur-ws.org/Vol-1235/paper-10.pdf

[14] T. Levendovszky, A. Dubey, W. Otte, D. Balasubramanian, A. Coglio, S. Nyako, W. Emfinger, P. Kumar, A. Gokhale, and G. Karsai, "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.

[15] P. Lopez, J. Medina, and J. Drake, "Real-time modelling of distributed component-based applications," in *Software Engineering and Advanced Applications, 2006. SEAA '06. 32nd EUROMICRO Conference on*, Aug 2006, pp. 92–99.

[16] J. Pasaje, M. Harbour, and J. Drake, "Mast real-time view: a graphic uml tool for modeling object-oriented real-time systems," in *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, Dec 2001, pp. 245–256.

[17] 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.

[18] ——, "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*, June 2009, pp. 26–33.

[19] B. Alpern and F. B. Schneider, "Verifying temporal properties without temporal logic," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 1, pp. 147–167, Jan. 1989. [Online]. Available: http://doi.acm.org/10.1145/59287.62028

[20] J. L. Medina and A. G. Cuesta, "From composable design models to schedulability analysis with uml and the uml profile for marte," *SIGBED Rev.*, vol. 8, no. 1, pp. 64–68, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1967021.1967030

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

[22] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee, "Incremental schedulability analysis of hierarchical real-time components," in *Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software*, ser. EMSOFT '06. New York, NY, USA: ACM, 2006, pp. 272–281. [Online]. Available: http://doi.acm.org/10.1145/1176887.1176927

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

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

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

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

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

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

[29] L. M. Kristensen, "State space methods for coloured petri nets," *DAIMI Report Series*, vol. 29, no. 546, 2000.

[30] K. Jensen, "An introduction to the practical use of coloured petri nets," in *Lectures on Petri Nets II: Applications*. Springer, 1998, pp. 237–292.

[31] K. Jensen, L. M. Kristensen, and L. Wells, "Coloured petri nets and cpn tools for modelling and validation of concurrent systems," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 3-4, pp. 213–254, 2007.

[32] S. Christensen, L. M. Kristensen, and T. Mailund, *A sweep-line method for state space exploration*. Springer, 2001.

[33] K. Jensen, "Condensed state spaces for symmetrical coloured petri nets," *Formal Methods in System Design*, vol. 9, no. 1-2, pp. 7–40, 1996.

[34] M. Westergaard, S. Evangelista, and L. M. Kristensen, "Asap: an extensible platform for state space analysis," in *Applications and Theory of Petri Nets*. Springer, 2009, pp. 303–312.

[35] S. Christensen, L. M. Kristensen, and T. Mailund, "A sweep-line method for state space exploration," in *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS 2001. London, UK, UK: Springer-Verlag, 2001, pp. 450–464. [Online]. Available: http://dl.acm.org/citation.cfm?id=646485.694466

[36] G. J. Holzmann, "An analysis of bitstate hashing," *Formal methods in system design*, vol. 13, no. 3, pp. 289–307, 1998.