Performance Estimation of Distributed Real-time Embedded Systems by Discrete Event Simulations*

Gabor Madl[1], Nikil Dutt[1], Sherif Abdelwahed[2]
[1] Center for Embedded Computer Systems
University of California, Irvine, CA 92697
[2] Department of Electrical and Computer Engineering
Mississippi State University, MS 39762, USA
[1]{gabe, dutt}@ics.uci.edu, [2]sherif@ece.msstate.edu

ABSTRACT

Key challenges in the performance estimation of distributed real-time embedded (DRE) systems include the systematic measurement of coverage by simulations, and the automated generation of directed test vectors. This paper investigates how DRE systems can be represented as discrete event systems (DES) in continuous time, and proposes an automated method for the performance evaluation of such systems. The proposed method also provides a way for the verification of dense time properties for a large class of DRE systems. This approach provides a formal executable model allowing to bridge the gap between simulations and formal verification. Our results show that the proposed DES-based evaluation method can achieve better coverage in large-scale DRE systems than alternative methods.

Categories and Subject Descriptors

I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; C.4 [**Computer Systems Organization**]: Performance of Systems – *Modeling techniques*

General Terms

Design, Performance, Verification

1. INTRODUCTION

Performance evaluation is a key challenge in the analysis of distributed real-time embedded (DRE) systems. Major design parameters that influence performance include real-time properties, such as task execution times and communication delays, the degree of parallelism in computations, and the throughput of the communication architecture.

Static schedulability methods [17, 11] provide guarantees for schedulability, but are often overly pessimistic when applied to event-driven systems. Moreover, they cannot model dynamic effects, such as varying communication delays and race conditions, as they do not capture the flow of data, and are less accurate than dynamic estimation techniques.

Synchronous languages [2] are another approach to specify timing constraints in globally synchronous and deterministic real-time embedded systems. Although the synchronous approach is popular in mission-critical systems, event-driven embedded systems often exhibit non-deterministic behaviors that synchronous languages cannot capture.

The current industry practice for the performance evaluation of large-scale DRE systems is mostly based on simulations. Common methods combine random simulations with a smaller set of directed test vectors to obtain the best possible coverage within a given time. However, the ad-hoc application of simulations prevents the systematic measurement of the coverage and quality of results.

Model checking provides alternative methods [7,8] for the dynamic analysis of event-driven distributed systems. Most symbolic representations – such as BDDs [4] – and temporal logic – such as LTL [22], and CTL [6] – focus on answering yes/no questions that turn performance evaluation into a tedious process. Also, there is often no way for tradeoffs between accuracy and scalability; if the exhaustive state space search is infeasible due to the large state space, the designer is left without a partial estimate of system performance.

This paper proposes a *discrete event simulation-based performance evaluation* method for DRE systems, that employ *fixed-priority scheduling*. We introduce a formal model for DRE systems based on discrete event scheduling [5] using the concept of *logical execution time* [10], and the *event order tree* shown in Section 4. Nodes in the event order tree represent events, and edges represent causality between the events. As events may arise non-deterministically, the tree may branch when different event orderings are possible. The proposed model explicitly captures the flow of data and communication effects (such as non-deterministic delays etc.) in event-driven systems for *dynamic performance evaluation.*

In the proposed approach we do not store *timed states*, like timed automata model checking methods, just events and constraints on the (global) timestamps of real-valued events. Note that this approach represents real-time properties in *continuous time.* Storing timed states is the most significant contributor to memory consumption in model checking tools. The proposed method has minimal memory requirements, providing a way for runtime on-the-fly analysis in adaptable DRE systems.

*This research was partially supported by the NSF grants CNS-0615438 and CNS-0613971

In this stage of development we do not address the termination problem, as we do not try to identify previously visited timed states, but use a constant horizon as a time limit for the analysis. There are model checking methods that do not have this limitation in theory, but in practice all model checking methods suffer from the termination problem due to the state space explosion problem. Our preliminary results show that the proposed DES-based evaluation method can achieve *better coverage in large-scale DRE systems* than alternative methods as shown in section 5. The abstract symbolic model allows *better simulation performance* compared to the actual simulations with comparable accuracy, providing an efficient method for design space exploration.

The organization of the paper is the following: Section 2 discusses related work; Section 3 presents the problem formulation and formalizes the model of computation (MoC) used for the analysis of DRE systems; Section 4 presents the DES-based performance evaluation and real-time verification method; Section 5 presents a large-scale avionics case study and performance comparisons to alternative methods; and Section 6 presents concluding remarks.

2. RELATED WORK

Static analysis methods: Scheduling theory is a widely used method for deciding the schedulability of DRE systems. Rate Monotonic Analysis (RMA) [17,11] and Earliest Deadline First (EDF) [17] methods both provide polynomial-time results with respect to the number of input tasks. SymTA/S [9] is a formal analysis tool that applies methods from scheduling theory for the performance analysis of complex heterogeneous multi-processor systems-on-chip (MPSoCs). A generic, component-based formal framework for the scheduling analysis and formal performance evaluation of platform-based embedded systems was proposed in [23]. Modular Performance Analysis [25] is an approach based on real-time calculus that models dependencies as arrival curves. Synchronous languages [2] provide a common mathematical domain for the analysis of globally synchronous systems with deterministic concurrency. Giotto [10] defines a timed-triggered language for the static analysis of schedulability in embedded systems.

Although all static analysis methods provide scalable solutions for performance evaluation they cannot model dynamic effects, such as varying delays and race conditions, as they do not capture the flow of data, and are less accurate than dynamic estimation methods. Communication in embedded systems is often non-deterministic, data-dependent, and hard to model as well-formed event streams.

In contrast, the method described in this paper is a formal analysis technique that captures dynamic effects, such as varying delays and race conditions in distributed systems, by explicitly modeling dependencies and the event-based triggering, and results in more accurate performance analysis at the price of being computationally more intensive.

Dynamic analysis methods: Simulations are the preferred and widely accepted way to evaluate DRE system designs in the industry today. A simulation-based design space exploration method, however, has several disadvantages. Developing the models for a design alternative may take weeks or months therefore only a handful of alternatives may be practically analyzed given the short product development cycles. Moreover, designers typically notice performance issues late in the design cycle - after the simulation model is complete - therefore addressing changes can be rather time-consuming and costly.

An approach for the formal model-based performance evaluation of DRE systems is demonstrated in Ptolemy II [16], that addresses some of these issues. Ptolemy II is a general modeling framework that composes heterogeneous models of computation to evaluate embedded systems, and also provides a way for performance evaluation through symbolic simulations of the formal MoCs. Ptolemy II focuses on deterministic models [15] and does not provide a systematic method for the measurement of state space coverage in non-deterministic models. Simulation-based performance evaluation is a popular approach for the evaluation of MPSoC designs as well. A semi-formal simulation-based performance evaluation method for MPSoCs was proposed in [12]. The authors represent execution traces as symbolic graphs for performance analysis, annotated with execution times obtained by simulating individual components of the system.

Although the approaches described in [12,16] improve simulation speed by utilizing symbolic representations of execution traces, the quality of results depends on the ad-hoc selection of test vectors. The method described in this paper builds on a symbolic simulation technique using DES to represent DRE systems. However, unlike the methods described in [12,16], we focus on non-deterministic systems with varying execution times, that we capture as intervals. In contrast to their approaches, our symbolic model captures all possible execution traces of the system, not just one execution trace. This is a more accurate model for event-driven DRE systems, where execution times are rarely constant. Moreover, we formalize our method for obtaining test vectors based on the DES model, that provide better coverage than random simulations.

Model checking methods: Model-checking provides alternative methods for the dynamic analysis of DRE systems. Several authors have proposed timed automata as a semantic domain for schedulability analysis [7, 8]. A timed automata-based approach for the thread-level analysis of DRE systems is presented in [24]. We have presented a formal method for deciding the schedulability of DRE systems based on timed automata in [20, 18, 19].

Although model checking provides the means for the formal real-time verification of event-driven DRE systems, its practical applicability to the performance evaluation of large-scale DRE systems is limited. The exhaustive verification of large-scale DRE systems is often infeasible in practice, and is often unnecessary, as the performance can usually be estimated accurately with less than perfect coverage. Moreover, most model checkers are based on logics [22, 6, 4] tailored towards yes/no questions which makes formal performance evaluation a tedious process.

In this paper we propose a method based on DES for the performance evaluation of large-scale DRE systems. Although the proposed method provides a way for the formal verification of real-time properties, fast symbolic simulations are its main advantage, and is directly applicable to large-scale systems, that cannot be analyzed using an exhaustive state space search. The DES-based analysis increases the coverage by gradually simulating the symbolic models, and can answer complex questions on the symbolic executable models. We show that this approach provides a way for fast design space exploration, and can achieve better coverage in some cases than alternative methods.

3. PROBLEM FORMULATION

In this section we define a formal model for event-driven DRE systems using fixed-priority scheduling. We utilize this model to define the formal performance evaluation problem using a continuous time model. We target asynchronous distributed event-driven systems, therefore we propose a *non-preemptive scheduling model*, that inherently captures varying communication delays as special "tasks" with execution intervals. The non-preemptive model allows to capture blocking waits and congestions in a formal setting.

3.1 Problem Elements

Our model of computation for event-driven DRE systems is a tuple $DRE = \{T, M, C, TR, D\}$ where:

- T is the set of *tasks*,
- M is the set of *machines*,
- C is the set of *communication channels*: $C \subseteq T$,
- TR is the set of *timers*: $TR \subseteq T$,
- D is the *task dependency relationship*: $D \subseteq T \times T$.

In our proposed DRE model of computation we have a set of machines $M = \{m_1, m_2, \ldots, m_q\}, q \in \mathbb{N}$, and a set of tasks $T = \{t_1, t_2, \ldots, t_n\}, n \in \mathbb{N}$ that we would like to execute on the machines. A task is an atomic non-preemptable unit that executes on a given machine. Each task has to be assigned to a machine and each machine can execute only one task at one time. Tasks are assigned to a specific machine by the mapping $\mathsf{machine}(t_k) : T \to M$. Each task t_k has an execution interval given by $[\mathsf{bcet}_k, \mathsf{wcet}_k]$, rather than a constant execution time, which is the major source of non-determinism in the models (other major sources of non-determinism are race conditions as described in Section 4.2). bcet_k is the *best case execution time* that denotes the shortest execution time, wcet_k is the *worst case execution time* that denotes the longest execution time of task (t_k).

Channels in set C can be viewed as special tasks that (1) buffer events as FIFOs, and (2) represent communication delays. They are not required in the models as tasks can exchange events directly, but provide a mechanism to reduce *blocking waits*. Blocking waits occur when the source task has to wait for the dependent task to return to the idle state. We introduce a (hypothetical) machine $m_c \in M$ that can execute an unbounded number of tasks at the same time. We express delays between tasks t_k and t_j, $\{t_k, t_j\} \in D$ by introducing a channel $c_d \in C$ that has an execution interval $[\mathsf{bcet}_d, \mathsf{wcet}_d]$ as specified by the delay, and we assign c_d to the machine m_c using the mapping $M(c_d) = m_c$, and add the dependencies $\{t_k, c_d\}, \{c_d, t_j\}$ to the set D. Note that we only map channels to the hypothetical machine in order to express delays as non-preemptive executions in a formal setting, we do not restrict the actual implementation of event channels. We call each task that is assigned to m_c a *channel*, and we refer to bcet_d as the *best case delay*, wcet_d as the *worst case delay*.

Timers in the set TR are special tasks that trigger the execution of tasks periodically as defined by their period. We give a more formal definition for timers in the next section by using the definition of events.

We define the set of dependencies $D = \{(t_a, t_b), (t_c, t_d), \ldots, (t_m, t_n)\}$. If the dependency (t_k, t_j) is part of the set D then task t_j has to execute after task t_k has finished. We say that task t_j *depends* on task t_k. A task may depend on several tasks and execute several times during the execution of the model.

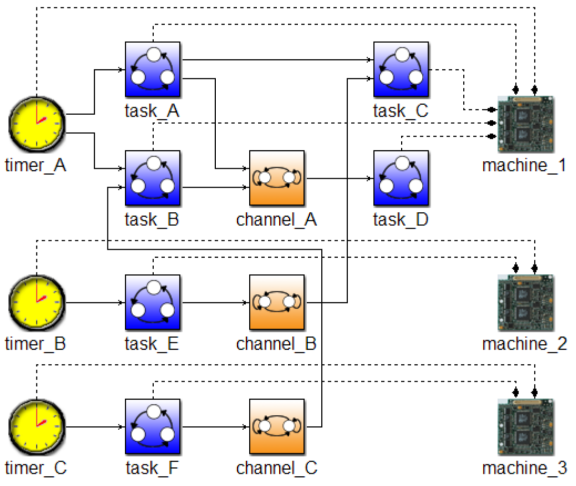

Figure 1: Example DRE Model

Note that when AND semantics are allowed, all parents need to finish their execution to trigger the execution of the (dependent) child task. This implies that events from parents that send events at a higher rate need to be continuously discarded to avoid buffer overflows. Therefore, at this stage of development, we only allow OR semantics for task dependency; any parent task can trigger the execution of the (dependent) child task. This restriction prevents buffer overflows in tasks that depend on parents, that send events with different rates. Moreover, we assume that there are no circular dependencies between tasks. Therefore, if task t_k depends on task t_j, then task t_j does not depend on task t_k. D is acyclic (a forest) with timers as root elements.

Figure 1 shows an example DRE model created using the GME tool [14]. Timers, tasks, channels, and machines are denoted by their respective icons in Figure 1. The solid arrows show the dependencies (set D) in the model. The mapping of tasks and timers to machines (the function $\mathsf{machine}(t_k) : T \to M$) is shown by the dashed arrows. All channels are mapped to m_c as defined above therefore we do not show their mapping in Figure 1.

3.2 Events

We represent the DRE MoC as an extension to discrete event systems in order to express execution intervals in continuous time. In DES, transitions depend only on the current state and the event label. In the DRE MoC, we define event labels as time stamped values from the domain of non-negative real numbers $(\mathbb{R}^+ \cup \{0\})$.

Timestamps model the (global) simulation time when the event has occurred. We define the function $\mathsf{time}(e) : E \to (\mathbb{R}^+ \cup \{0\})$ to return the timestamp value of event $e \in E$ where E is the set of all (infinite) events generated by the system.

In the DRE MoC, tasks receive a potentially infinite sequence of events (timestamped values as event labels) in chronological order. The task then outputs a timestamped event for each input event. We denote the sequence of input events of a task t_k as $I_k = \{i_{k0}, i_{k1}, \ldots\}$, the sequence of output events as $O_k = \{o_{k0}, o_{k1}, \ldots\}$, $t \in T, i_{k0}, i_{k1}, \cdots \in E, o_{k0}, o_{k1}, \cdots \in E$. The order of events in the output sequence of each task is the same as the order of events in the input sequence of the task, o_{k0} is the response for i_{k0}, o_{k1} is the response for i_{k1}, etc. The timestamps of input events must be smaller than or equal to their corresponding output events. We formalize this constraint as follows: $(\forall t_k \in T)(\forall i_k \in I_k)(\forall o_k \in O_k)$ $(\mathsf{time}\ (i_k) \leq \mathsf{time}\ (o_k))$. Note that the discrete event simulation is completely deterministic if timestamps are unique constants.

Each task can process only one event at a time, for each input event $i_{kx} \in I_k$, the corresponding output event $o_{kx} \in O_k$ has to be generated before the task can receive its next input event $i_{ky} \in I_k$. Channels in the set C provide FIFO buffers to store events that cannot be immediately processed by their target tasks.

The set TR denotes a special class of tasks called *timers.* A timer $tr \in TR$ is a task that generates output events such that for the timestamp of any two consecutive events $(o_{tr}, o'_{tr} \in E)$ $\mathsf{time}(o'_{tr}) - \mathsf{time}(o_{tr}) = \mathsf{period}_{tr}$. We define the period of the timer as $\mathsf{period}_{tr} = \mathsf{bcet}_{tr} = \mathsf{wcet}_{tr}$.

To distinguish the special classes of tasks from non-special tasks we refer to the set $T-C-TR$ as the set of *computation tasks.* Computation tasks model actual tasks being executed on the machines.

Computation tasks, timers, and event channels in the DRE MoC can be composed using events; the output event of a task may serve as an input event to another task (tasks). In these cases the same event label triggers multiple transitions. We make the restrictions that timers cannot have input events and the output events of event channel can only be inputs to computation tasks. Moreover, the event flow has to satisfy the dependencies in set D; if the output of a task $t_1 \in T$ is the input of task $t_2 \in T$ then the dependency (t_1, t_2) has to be present in set D. Similarly, if a dependency (t_a, t_b) is present in set D then the output event of t_a has to be the input event of t_b.

3.3 Task States, Schedulers

We define three states for each computation task; `init`, `wait`, and `run`. Whenever a task receives an event from another task (including event channels and timers) the transition from the `init` state to the `wait` state is triggered. We refer to tasks in the `wait` state as *enabled tasks.* Enabled tasks are ready to execute.

We model scheduling policies by utilizing priorities. We model schedulers as discrete event systems that compose with tasks using events. Our model for schedulers keeps track of enabled tasks by putting them in an execution queue. Whenever the execution queue is non-empty the scheduler chooses a task (or possibly several tasks) for execution by generating an event triggering the transition from the `wait` state to the `run` state in the selected task(s). We assume that tasks can distinguish between events coming from tasks and schedulers in the DRE MoC. Events generated by schedulers are also labeled as timestamps.

Figure 2 shows a possible DES representation of the DRE example shown in Figure 1. We create a finite state machine model for each task, channel, timer, and scheduler, that compose using events. We denote input events as $e?$, output events as $e!$. If the output event of a transition is the input event of another transition then the two transitions are synchronized; their events must have the same timestamps when the transitions are taken. If two (or more) transitions are synchronized either all of them has to be taken, or none

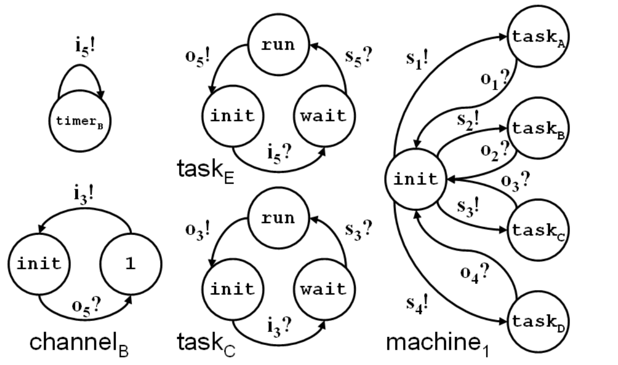

Figure 2: Composing DES Models using Events - Partial Representation of the DRE Example Shown in Figure 1

of them. We model scheduling policies by introducing priorities between transitions. For example, a simple fixed priority scheduling policy between tasks t_a, t_b, t_c, and t_d may be implemented by introducing priorities between the transitions marked as $s_1!, s_2!, s_3!, s_4!$. The priorities may enforce a fixed execution order between enabled tasks (a task is enabled if it is in the `wait` state).

All schedulers in the DRE MoC implement a non-preemptive scheduling policy; the transition from the `run` state to the `init` state is triggered when the task generates an output event, and we do not allow transition from the `run` state to the `wait` state. Each computation task $t_k \in (T - C - TR)$ generates an output event with a timestamp between $[\mathsf{bcet}_k + \mathsf{time}(s_k),\ \mathsf{wcet}_k + \mathsf{time}(s_k)]$ where s_k is the event generated by the scheduler that triggered the transition from state `wait` to state `run` in task t_k.

In the DRE MoC every task has to be mapped into a machine. As a machine is a model for a single node in a distributed system each machine has its own scheduler with its own execution queue, and each scheduler might implement a different scheduling policy.

3.4 Performance Evaluation Problem

We now utilize the DRE MoC to define the performance evaluation problem and schedulability problem for non-preemptive DRE systems. We use the notation for the sequence of input events of a task t_k as $I_k = \{i_{k0}, i_{k1}, \dots\}$, the sequence of output events as $O_k = \{o_{k0}, o_{k1}, \dots\}$, $t \in T, i_{k0}, i_{k1}, \dots \in E, o_{k0}, o_{k1}, \dots \in E$. We specify deadlines for each computation task t_k using the mapping $d_k : T \rightarrow \mathbb{N}$. Deadlines for each task and each input event are counted from the timestamp of the input (enabling) event $(\mathsf{time}(\mathrm{i}_k))$.

Definition 1. **(Schedulability):** *A computation task $t_k \in T$ is schedulable if it always finishes its execution before its respective deadline. The DRE MoC is schedulable if all tasks are schedulable. We formalize this condition using the DRE MoC as follows:* $(\forall t_k \in (T-C-TR))(\forall i_k \in I_k)(\forall o_k \in O_k)$ $\mathsf{time}(o_k) < \mathsf{time}(i_k) + d_k$.

Definition 2. **(Run):** *A* run *or* execution trace *of the DRE MoC is the chronological sequence of events occured in the model. A run is* valid *if it is schedulable, that is if for all input (i_k) and their corresponding output (o_k) events in the execution trace* $\mathsf{time}(o_k) < \mathsf{time}(i_k) + d_k$*, otherwise it is* invalid.

Definition 3. **(End-to-end computation time):** *We define the end-to-end computation time between an input event i_{jn} of task t_j and an output event o_{km} of task t_k as the maximum possible difference between the events' timestamps along all the possible runs of the model* $\text{end-to-end}(o_{km}, i_{jn}) = \max[\text{time}(o_{km}) - \text{time}(i_{jn})]$, *if* $\exists\{(t_j, t_a), (t_a, t_b), ..., (t_b, t_j)\} \in D$. *If task t_k does not depend on task t_j in the DRE MoC ($\nexists\{(t_j, t_a), (t_a, t_b), ..., (t_b, t_j)\} \in D$), we define* $\text{end-to-end}(o_{km}, i_{jn}) = \infty$.

It is a well-known fact that the sum of local worst case execution times does not necessarily result in worst case end-to-end computation times. We now demonstrate this problem in non-preemptive DRE systems to motivate our approach for formal performance evaluation using the simple DRE example shown in Figure 1. We define the period of each timer to be 100 time units, and the delay of each channel to be 0.

Figure 3 illustrates the first period of DRE model execution traces shown in Figure 1 using the parameters in Table 1. In this example, for most tasks the bcet_k time equals the wcet_k time to reduce complexity, for easier illustration. We utilize fixed-priority scheduling in `machine_1` between tasks t_A, t_B, t_C, and t_D. Tasks t_E and t_F are executed concurrently and have their own schedulers. Note that EDF scheduling would result in a deadline miss by task t_B, as it scheduled the sequence t_A, t_C, t_B. This illustrates that EDF is not optimal in the non-preemptive DRE MoC.

The execution trace in the left of Figure 3 demonstrates that the system is schedulable if all tasks execute using their `wcet`. The trace in the middle of Figure 3 shows that the system is schedulable when `bcet` are considered during the execution trace. However, the trace in the right of Figure 3 shows that task t_C might miss its deadline if task t_E executes for 71 time units. This example shows that the performance evaluation of event-driven non-preemptive DRE systems has to consider *execution intervals* rather than worst case execution times, and justifies the need for automated response time analysis.

4. PERFORMANCE ESTIMATION OF DRE SYSTEMS BY DES

This section describes the proposed DES-based performance evaluation method for event-driven DRE systems expressed using the DRE MoC. We introduce the *event order tree* and show how it can be utilized for performance estimation.

4.1 Event Order Tree

Definition 4. **(Equivalent execution traces):** *In the DRE MoC two execution traces are equivalent, if the two execution traces contain the same events, and the chronological order of events in both execution traces is the same.*

Note that for equivalence only the order of events have to be the same, not the timestamps of events (untimed equivalence). We propose a directed tree representation for the valid traces of a DRE model, called *event order tree*. Each node in the event order tree represents an event and the (global) time constraint on the current event's timestamp. The path from the root of the event order tree to a node represents possibly infinite number of equivalent execution traces of a DRE model. There is a directed edge from node

Task	t_A	t_B	t_C	t_D	t_E	t_F
bcet	10	10	10	10	50	70
wcet	10	10	10	10	90	70
deadline	22	25	12	32	100	100

Table 1: Timing Information for the DRE Model Shown in Figure 1

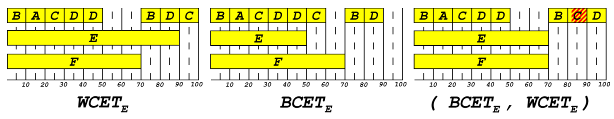

Figure 3: Execution Traces of the DRE Model Shown in Figure 1

A to node B if event B may be raised after event A, and there are no other events between them.

Figure 4 shows the event order tree for the DRE model shown in Figure 1 (thicker borders explained in Section 5). Computations in the model are triggered by the timers, that generate events i_1, i_2, i_5, i_6, therefore we label the root as $\mathbf{i_1 i_3 i_5 i_6}$. All these events are generated at (global) time 0, therefore we represent the constraint on their timestamps as **[0, 0]** in the root. The schedulers trigger the execution of tasks t_B, t_E and t_F by generating the s_2, s_5, s_6 events. We label the immediate child of the root in the event order tree as $s_2 s_5 s_6$. Scheduling tasks for execution after they become enabled is instantaneous in our discrete event scheduler, therefore the time constraint on the timestamps of events $s_2 s_5 s_6$ remains $[0, 0]$. Although the time constraints in the root and the node marked as $s_2 s_5 s_6$ are identical, there is a causal ordering between them in the DES model; a task can only be scheduled for execution after it has received an input event. The causal orderings between events with the same timestamp correspond to zero-time transitions in other MoCs, such as timed automata.

Each path in the event order tree from the root to the leaves imposes constraints on the execution intervals of tasks by defining constraints on the timestamps of events. For example, the path from the root to the leftmost leaf in the same tree requires task t_E to finish its execution after task t_D has finished its execution, and before task t_F finishes it execution in the $(50, 70)$ interval as shown by the constraint $\mathbf{o_5 i_3 (50,\ 70)}$ in the event order tree. Therefore, the leftmost path in the event order tree shown in Figure 4 restricts the execution time of task t_E to $(50, 70)$.

Definition 5. **(Branching intervals):** *We refer to intervals implied by equivalent execution traces as* branching intervals. *Branching intervals are always subsets of the execution intervals* $[\text{bcet}_k,\ \text{wcet}_k]$ *of tasks.*

For example, in the case of task t_E its three branching intervals are: $[50, 50], (50, 70), [70, 70] \subset [50, 70]$, as shown by the constraints $\mathbf{o_4 o_5 i_3 [50,\ 50]}$, $\mathbf{o_5 i_3 (50,\ 70)}$, $\mathbf{o_5 o_6 i_2 i_3 [70,\ 70]}$ in the event order tree.

4.2 Branches in the Event Order Tree

The execution of the DRE model on `machine_1` is deterministic, tasks execute in the order t_B, t_A, t_C, t_D, t_D, while tasks t_E and t_F execute on `machine_2` and `machine_3` in parallel. We reach the first non-deterministic choice in the ordering of events after task t_D starts executing for the second time within the period; if task t_E executes for its `bcet`

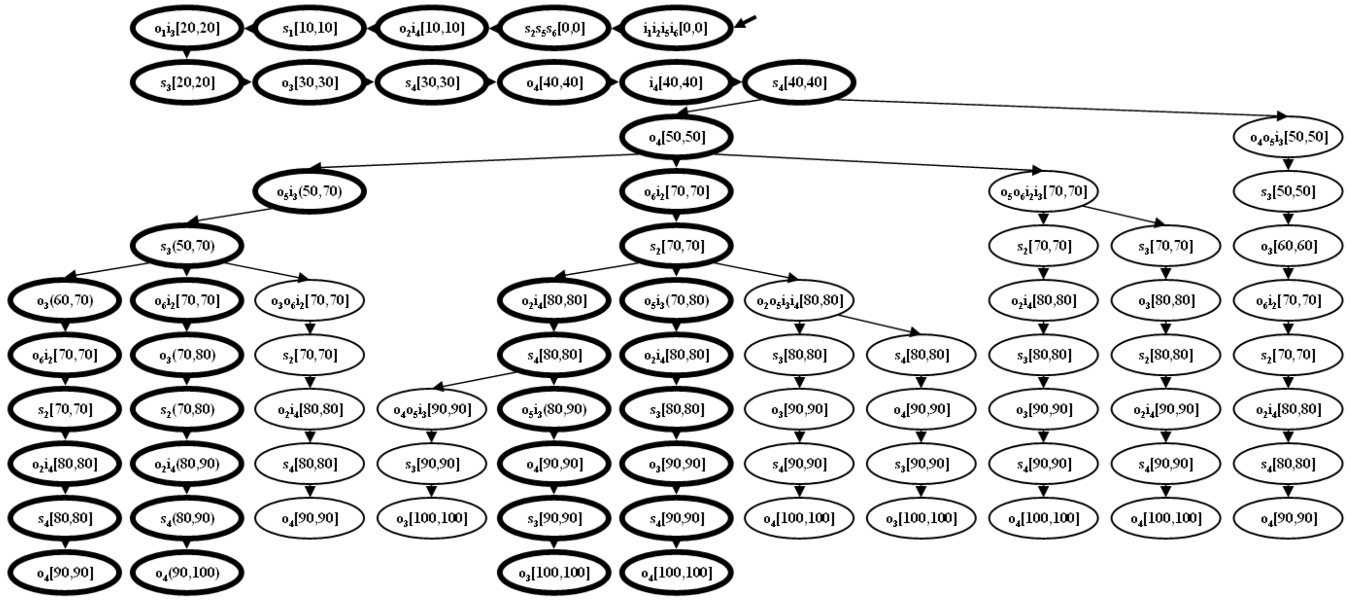

Figure 4: The Event Order Tree of the DRE Example in Figure 1 using the Parameters in Table 1

then tasks t_D and t_E finish their execution simultaneously (constraint $\mathbf{o}_4\mathbf{o}_5\mathbf{i}_3[\mathbf{50,\ 50}]$); otherwise task t_D finishes its execution first (constraint $\mathbf{o}_4[\mathbf{50,\ 50}]$), and then task t_E finishes its execution. To capture this non-determinism the event order tree has to branch at node $\mathbf{s}_4[\mathbf{40,\ 40}]$. In the DRE MoC we also consider race conditions between tasks assigned to the same machine only if they receive events from tasks assigned to other machines.

Definition 6. **(Race condition):** *If for two tasks* $t_k, t_j \in T$, $\mathsf{machine}(t_k) = \mathsf{machine}(t_j)(\exists i_k \in I_k, i_j \in I_j)(\mathsf{time}(i_k) = \mathsf{time}(i_j))$, *and* $(\mathsf{machine}(t_s) \neq \mathsf{machine}(t_k) \vee \mathsf{machine}(t_r) \neq \mathsf{machine}(t_j), t_s \in T, t_r \in T, \{t_s, t_k\} \in D, \{t_r, t_j\} \in D$ *then there is a* race condition *between task* t_k *and* t_j*.*

For example, if tasks t_E and t_F finish at the same time there is a race condition between tasks t_B and t_C. We identify race conditions the following way: whenever a set of tasks receives events with the same timestamp we check whether the tasks that generated that event are assigned to the same machine as the set of tasks. If not, race conditions may be present. If race conditions are present between a set of tasks we have to consider each task to receive its respective event first, therefore in these cases the event order tree has to branch for each task.

Consider the node $\mathbf{o}_5\mathbf{o}_6\mathbf{i}_2\mathbf{i}_3[\mathbf{70,\ 70}]$ in the event order tree shown in Figure 4. This node represents the execution trace where tasks t_B, t_A, t_C, t_D, t_D execute in this order in `machine_1`, and tasks t_E and t_F finish their execution at the same time. The event order tree branches and we consider both the case when task t_B receives its start even first ($\mathbf{s}_2$), or when task t_C receives its start event first ($\mathbf{s}_3$) due to race conditions.

Definition 7. **(Branching point):** *We refer to nodes in the tree, where the event order tree branches due to non-deterministic execution times, or race conditions as* branching points.

4.3 Real-time Verification by DES

In this subsection we propose a method for the real-time verification of a large class of DRE models using the event order tree. The event order tree is a symbolic representation of all distinguishable execution traces of the DRE model from a timing perspective as we show in this section. We build on the results of this section to propose a method for the on-the-fly construction of the event order tree in Subsection 4.4, providing a way for formal performance evaluation with a systematic measurement of state space coverage.

Timers in the DRE MoC introduce periodicity in the models. Since the DRE MoC allows the use of multiple timers with different periods we need to find the least common multiplier of timer periods, which we refer to as *time limit.* We make the restriction that all tasks have to be in the `init` state when events timestamped with the time limit are generated.

$$\forall(t_k \in T - C - TR)\ \mathsf{state}(t_k, time\_limit) = \mathtt{init} \qquad (1)$$

This restriction is sufficient, but not necessary for a schedulable DRE model, as in pipeline architectures the processing of older events may overlap with the processing of newer events at different stages in the pipeline, therefore we may not reach a condition where all tasks return to the `idle` state at once. In pipelined systems we can either verify the system to a limited horizon – which does not guarantee that the system will work properly after the time limit – or use other model checking techniques on the DRE MoC, such as timed automata, as described in [20, 18, 19]. In the rest of this section we show that if Equation 1 is satisfied, we can verify DRE systems by exhaustively enumerating all the execution traces corresponding to all paths in the event order tree from the root to the leaves.

Theorem 1. **(Repeatable property):** *The event order tree of a given DRE model repeats itself from all its leaves. We refer to this property of event order trees as* repeatable.

Proof (outline): We build on Equation 1 that tasks have to be in their initial states (`init`) when the time limit is reached, that is the least common multiplier of timer periods. The timers generate the same events that have appeared in the root, with the timestamps of the time limit. Since there is no relative difference between the timestamps of events generated by the timers the DRE model can exhibit the same execution traces as before. □

We only build the event order tree until we reach the time limit on the timestamps of events. For example, the event order tree shown in Figure 4 repeats itself from all leaves. It is important to note that even though a DRE system may utilize several timers with different periods there is only one event order tree, rather than a forest. New events generated by faster timers are considered as branching points or are simply appended to the leaves if no tasks are running when the timer generates a new event.

THEOREM 2. **(Finite number of nodes):** *There is a finite number of nodes in the event order tree.*

Proof (outline): In the DRE MoC a finite number of events are generated with timestamps within any interval, because (1) we only consider the boundaries of intervals, (2) timers generate events at discrete time steps, and (3) each task generates finite number of output events for each input event. Therefore, each branch has a finite number of children in the event order tree. There is a finite number of branches – at most one for each executing tasks, and one for each enabled task that may be in race conditions. Since there is a finite number of branches and each branch has a finite number of children, the event order tree has a finite number of nodes. □

THEOREM 3. **(Worst case execution trace):** *We define the* worst case execution trace *of equivalent execution traces as the execution trace where tasks produce output events with maximum value timestamps from their branching intervals (as introduced in Definition 5). If the worst case execution trace of equivalent execution traces is valid, then all equivalent execution traces are valid.*

Proof (outline): The order of events in equivalent execution traces is fixed. Therefore, none of the tasks is forced to wait for longer when the execution times of some tasks are decreased, than when execution times are left unchanged. None of the tasks generate events with timestamps larger than in the worst case execution trace, otherwise the ordering of events would change. If the worst case execution trace is valid, then all equivalent traces are valid, since tasks within those execution traces generate events with timestamps less than or equal to the worst case execution trace, therefore they do not violate their deadlines. □

We have shown that the event order tree has a finite number of leaves, therefore DRE models have finite number of equivalent execution traces. We have also shown that the real-time properties of equivalent execution traces can be verified using a single discrete event simulation. The set of paths in the event order tree from the root to the leaves gives all the possible equivalent execution traces of a DRE model. The exhaustive discrete event simulation of all the possible equivalent execution traces in the event order tree of a DRE model consists of a finite number of discrete event simulations, therefore it is a valid method for the real-time verification of DRE models that satisfy Equation 1.

4.4 On-the-fly Detection of Branching Points in the Event Order Tree

This subsection describes how we can detect branching points at runtime, providing a way for the on-the-fly construction of the event order tree. By enumerating all execution traces corresponding to the paths from the root of the event order tree to the leaves, we can estimate the system's performance with 100% coverage. However, the exhaustive analysis of large-scale DRE systems is most often infeasible in practice due to the state space explosion problem. Therefore, in most practical scenarios we cannot build the whole event order tree in advance due to storage constraints, and we can only enumerate some paths of the event order tree due to time constraints. In these cases we obtain results using a partial state space search, and therefore we cannot guarantee their correctness. We can, however, achieve better coverage and confidence than with the existing methods, as shown in Section 5.

There are two major ways for building and analyzing the event order tree. The first option is to build the event order tree in a breadth-first search (BFS) fashion. The BFS-based approach stores the event order tree in the memory, and iteratively build the tree from the leaf-candidates. This approach requires that we store timing information (and times states) for all leaf-candidates of the event order tree in order to quickly restore the timed state of the system corresponding to the actual leaf-candidates, and check for deadlines and end-to-end computation times. The BFS-based approach has significant memory overhead, and resembles an exhaustive model checking method.

In this paper we propose a depth-first search (DFS)-based approach to obtain the event order tree. The DFS-based approach has minimal memory overhead, as it does not store the event order tree in the memory. We detect branching points in the event order tree during simulation traces, and then use this information to direct the discrete event scheduler to iteratively explore unique paths in the event order tree.

Note that although there are several model checkers that implement a BFS-based or/and a DFS-based search algorithm to enumerate symbolic state spaces, they are optimized to check simple properties using some logic - such as LTL [22] or CTL [6]. For the evaluation of embedded systems designers often want to find the maximum/minimum value of certain design parameters, or simply check whether the system performance gets better or worse when they change a design parameter. These conditions often require multiple model checker runs in order to translate these properties into a set of yes/no questions, which becomes impractical, cumbersome, and time consuming.

The key problem that we need to address is to detect branching points at runtime, and then exploit this information to construct new directed simulation traces in the future that enumerate traces representing unique branching intervals. We now describe a simple and practical approach to address this problem.

In our implementation all events are globally observable, and *each task detects its own branching intervals.* As we discussed in Definition 5, branching intervals are always subsets of the execution intervals [bcet_k, wcet_k] of tasks, since the order of events in an execution trace can only change if an event is raised earlier/later than another event. Moreover, all branching intervals represent different orderings of

Algorithm 1 Obtaining and Enumerating the Event Order Tree by Discrete Event Simulations

```
1: create the superset of race conditions $R$
2: set the execution time for all tasks $t_k \in T$ to their $\mathsf{wcet}_k$
   time, and the next execution time for all tasks to their
   $\mathsf{bcet}_k$ time, respectively ($\forall t_k \in T$ $\mathsf{exec\_time}_k = \mathsf{wcet}_k$,
   $\mathsf{next\_time}_k = \mathsf{bcet}_k$)
3: // enumerate all branching intervals
4: for all permutations of $\mathsf{exec\_time}_k$ assignments, obtained using the $\mathsf{next\_time}_k$ variables do
5:    clear the superset $R$
6:    call discrete_event_simulation () described in Algorithm 2
7:    // enumerate all race conditions with the current $\mathsf{exec\_time}_k$ assignments
8:    for all permutations of events in superset $R$ do
9:       call discrete_event_simulation () described in Algorithm 2
10:   end for
11: end for
```

events, therefore we only need to consider events within the [bcet_k, wcet_k] intervals of tasks to detect all branching points. Also, as described in Section 4.2, we consider race conditions in the models as well. During a single execution trace, whenever we encounter a race condition between events using Definition 6, we add the events in a set, and then add the set to the superset containing all sets of race conditions. For each event in a set we need to consider the possibility that it is executed first due to a race condition, and we need to consider all permutations between the sets in the superset. *We detect all events on-the-fly during simulations, and check all their possible permutations at the boundaries of branching intervals*, therefore we enumerate all the paths in the event order tree. Algorithm 1 describes our algorithm for generating all permutations of events by iterative simulations. Note that we do not set all tasks' execution times to their next_exec time simultaneously, rather we generate all permutations.

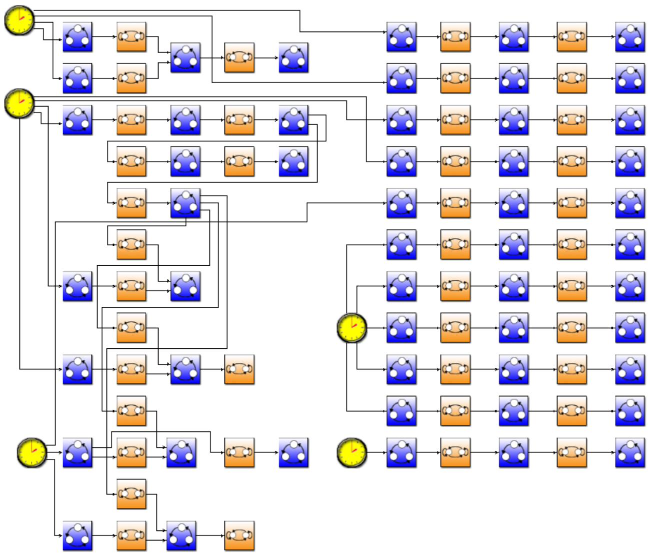
Figure 5: Example DRE System Case Study

Algorithm 2 function discrete_event_simulation ()

```
1: run directed discrete event simulation, during which
   each task stores its start time as $\mathsf{start}(_k)$
2: during the simulation all tasks $t_k$ observe events $e_i$ that
   are raised in the [$\mathsf{start}(_k) + \mathsf{bcet}_k$, $\mathsf{start}(_k) + \mathsf{exec\_time}_k$)
   interval
3: if $\mathsf{start}_k + \mathsf{next\_time}_k < \mathsf{time}(e_i)$ then
4:    // we have encountered a branching point in the ($\mathsf{bcet}_k$, $\mathsf{wcet}_k$) interval
5:    record $\mathsf{time}(e_i)$ - $\mathsf{start}_k$ in $\mathsf{next\_time}_k$
6: else
7:    do nothing, event will be considered in subsequent simulations
8: end if
9: // find all race conditions with the current $\mathsf{exec\_time}_k$ assignments
10: for all race conditions detected between events
    $e_i, e_j, \ldots e_k$ during the simulation do
11:    search for the set containing events $e_i, e_j, \ldots e_k$ in superset $R$
12:    if the set is found then
13:       do nothing
14:    else
15:       add the set $S = \{e_i, e_j, \ldots e_k\}$ to the superset $R$
16:    end if
17: end for
```

5. PRACTICAL APPLICATION AND PERFORMANCE

In this section we evaluate the proposed DES-based performance estimation method as implemented in the open-source DREAM tool [21]. Figure 5 shows the case study used for the performance estimation. The DRE model is loosely based on a real-time CORBA avionics application implemented in the Boeing Bold Stroke execution framework. The model consists of 98 tasks (including FIFO channels and timers) and 57 dependencies between tasks. Due to space constraints we can only illustrate the size and design of the model used for the analysis. Our approach for the modeling of the Bold Stroke framework is described in [20]. The case study shown in Figure 5 is described in [21], and is distributed with the open-source DREAM tool.

As described in Section 2, existing methods for real-time analysis and performance estimation have limited use in non-preemptive event-driven asynchronous systems. Static schedulability methods are not directly applicable to this scheduling model as demonstrated in Section 3.4, and are often overly conservative, limiting the accuracy of performance estimation results. Simulations capture dynamic effects in DRE systems, providing better accuracy, but the coverage of simulations is hard to measure. Model checking, on the other hand, provides a way for exhaustive analysis, but the abstractions required to prevent the state space explosion problem often result in decreased accuracy. The proposed DES-based performance estimation method combines model checking with simulations, therefore we compare the DES-based performance estimation results to random simulations, as implemented in the DREAM tool, and to timed automata model checking methods implemented in the UPPAAL model checker [13], and the Verimag IF toolset [3].

We focused on the analysis of two properties in our experiments; (1) we measured the *end-to-end computation time* of the application, as defined in Definition 3, from the first event generated by the timers, till the time when all tasks have finished their execution, and (2) we performed *schedulability analysis*, by formally analyzing whether any of the tasks may violate their deadlines.

5.1 Comparison with Random Simulations

The main advantage of the DES-based method is that it gradually increases coverage over time. Random simulation-based methods do not have this property. Random simulations assign execution times following a uniform distribution from the $[\mathsf{bcet}_k, \mathsf{wcet}_k]$ intervals of tasks. Let's denote the two endpoints of a branching interval as l_{k_i}, h_{k_i}, where l_{k_i} refers to the lower bound on the branching interval, and h_{k_i} refers to the higher bound on the branching interval $\mathsf{bcet}_k \leq l_{k_i} \leq h_{k_i} \leq \mathsf{wcet}_k$. Then we can formalize the probability that the random execution time is within the branching interval as follows:

$$P = \frac{h_{k_i} - l_{k_i}}{\mathsf{wcet}_k - \mathsf{bcet}_k} \tag{2}$$

Note that $\lim_{(h_{k_i} - l_{k_i}) \to 0} P = 0$, therefore the probability that an exact number is chosen randomly from a continuous-time interval is close to 0, even if we execute infinite number of simulations. Also, the smaller the branching interval, the less chance that we actually consider it during simulation. Since there is a higher chance that the execution time is picked from larger branching intervals, repetitive simulations will pick execution times from branching intervals that have already been chosen for simulation. To illustrate this problem, consider the event order tree as shown in Figure 4. The nodes with the thinner borders correspond to execution traces, that represent race conditions, and cases when two (or more) events are released with the same timestamp. We see that these cases represent the majority of possible unique orderings of events in this simple example. In larger systems we can expect even worse results, as the number of branching intervals and race conditions may grow exponentially with respect to the number of tasks in the system.

As we have seen from Equation 2, the chance to find these execution traces using random simulations is close to 0. However, in the actual system the execution times rarely follow a uniform distribution; it is quite probable that some execution times are more frequent than others, and that the real system encounters execution traces that were not considered during the simulation-based evaluation process. Since these traces are not simulated, designers will also fail to recognize how the system performance/schedulability might change due to dynamic effects such as race conditions or congestions. Therefore, we conclude that random simulations may be useful for the first steps of performance evaluation, but can achieve only partial coverage of the possible execution traces over time.

In contrast, the method presented in this paper gradually increases coverage over time. Moreover, we consider each branching interval only once, and we check the worst case times directly, rather than a random number from the branching interval. Therefore, the proposed DES-based method can discover significantly more corner cases than random simulation-based performance estimation techniques.

To check whether our observations are relevant in large-scale systems, we ran experiments to compare random simulations and the DES-based method on the model shown in Figure 5. We used a 1.7GHz Pentium 4-M machine for the tests with 1GB Memory, running Linux kernel 2.6.20. On this test configuration, the Dream 0.7 Beta release can simulate one execution trace of the DRE case study shown in Figure 5 in $\sim$30 ms. The fast performance is the result of the symbolic DES-based representation. We ran both random simulations and the DES-based method on the model shown in Figure 5 for a week. We used the open-source Dream tool for the random simulations as well, therefore all improvements in the DES-based analysis are the result of the better state space coverage. We were able to simulate $\sim$20 million (2×10^7) non-equivalent execution traces (the execution order of tasks is different) of the case study using the DES-based method in one week. This coverage can only be achieved using model checking techniques within this short time. Our experiments show that the DES-based analysis can obtain higher bounds on the worst case end-to-end performance than random simulations [21].

The difference comes from the fact that the DES-based method has better state space coverage, and therefore it is more accurate for performance estimation than random simulations. Even though the DES-based method cannot always obtain the highest bounds on the end-to-end performance, the combination of model checking and directed simulations along the execution tree provided the best coverage that we could achieve within a week on this case study. This shows that the proposed DES-based verification method is practically applicable for the performance evaluation of large-scale systems.

5.2 Comparison with Timed Automata Model Checking Methods

We have used Dream to generate timed automata representation from DRE models as described in [20]. Uppaal and the IF toolset are two leading model checkers for real-time verification with several years of development history. Although both Uppaal and IF build on the timed automata model of computation they are inherently different. Uppaal uses a traditional timed automata model [1] extended with integer-valued variables, IF, on the other hand, uses transition priorities to express time constraints.

We have not conducted extensive comparisons between Dream and timed automata model checkers yet to reach meaningful conclusions on how their verification performance compares in general. In the case studies that we've analyzed, timed automata model checkers usually perform better than the proposed DES-based method, on smaller models, that have a high degree of non-determinism. Earlier we have successfully used Uppaal for the real-time verification of DRE systems consisting of $\sim$30 tasks/event channels as described in [20]. Timed automata model checkers employ symbolic state representations that allow for efficient heuristics and compact state space representation. Therefore, both Uppaal and IF implement memory-bounded model checking.

On large-scale models, such as the case study shown in Figure 5, however, both timed automata model checkers run out of memory, and are unable to give partial results to designers. Although Dream does not run out of memory on these examples, the verification time increases exponentially. The impact of this problem could be potentially reduced by implementing the model checker on a distributed platform. In our experience, timed automata model checkers are useful for the performance evaluation of DRE systems that can be modeled with less than ∼60-100 clocks (occasionally better on mostly deterministic models), but cannot be applied for the performance estimation of large-scale systems in practice. Moreover, since performance estimation has to be formalized as a yes/no question, designers have to "guess" what a close bound on the end-to-end computation time could be, and check whether the performance is smaller or not. Our experiences have shown that timed automata model checkers are well-suited for the real-time verification of small/medium size systems, but their practical application for the performance estimation of large-scale DRE systems is limited, and cannot be compared to the proposed DES-based method – or even random simulations – on large-scale systems, due to the state space explosion problem as a result of the exhaustive analysis.

6. CONCLUDING REMARKS

This paper presents an approach to model DRE systems as discrete event systems using a continuous-time model, and proposes a method for formal performance evaluation and real-time verification. The proposed method explicitly captures the data flow, and models communication and execution intervals using a non-preemptive scheduling model. The DRE MoC provides a formal executable model allowing to bridge the gap between simulations and formal verification. Our benchmarks based on a large-scale avionics case study show that the DES-based performance evaluation method can achieve better coverage than alternative methods, and provides a way for the systematic measurement of coverage. The DES-based performance estimation and verification method has been implemented in the open-source Dream tool available at `http://dre.sourceforge.net`.

7. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The Synchronous Languages 12 Years Later. *Proceedings of the IEEE*, 91:64–83, 2003.

[3] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF Toolset. *Formal Methods for the Design of Real-Time Systems, LNCS 3185*, pages 237–267, 2004.

[4] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[5] C. G. Cassandras. *Discrete Event Systems, Modeling and Performance Analysis*. Irwin, 1993.

[6] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. *Logic of Programs, LNCS*, 131:52–71, 1981.

[7] C. Ericsson, A. Wall, and W. Yi. Timed Automata as Task Models for Event-Driven Systems. In *Proceedings of RTSCA*, 1999.

[8] T. Gerdsmeier and R. Cardell-Oliver. Analysis of Scheduling Behaviour using Generic Timed Automata. *Electronic Notes in Theoretical Computer Science*, 42, 2001.

[9] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis - the SymTA/S Approach. *IEE Proceedings Computers and Digital Techniques*, 152:148–166, 2005.

[10] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.

[11] M. H. Klein, T. Ralya, B. Pollak, and R. Obenza. *A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[12] K. Lahiri, A. Raghunathan, and S. Dey. System-Level Performance Analysis for Designing On-Chip Communication Architectures. *IEEE Transactions on Computer Aided-Design of Integrated Circuits and Systems*, 20:768–783, 2001.

[13] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[14] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, and J. Sprinkle. Composing Domain-Specific Design Environments. *Computer*, pages 44–51, 2001.

[15] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5), 2006.

[16] E. A. Lee, C. Hylands, J. Janneck, J. D. II, J. Liu, X. Liu, S. Neuendorffer, S. S. M. Stewart, K. Vissers, and P. Whitaker. Overview of the Ptolemy Project. Technical Report UCB/ERL M01/11, EECS Department, University of California, Berkeley, 2001.

[17] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, 1973.

[18] G. Madl and S. Abdelwahed. Model-based Analysis of Distributed Real-time Embedded System Composition. In *Proceedings of EMSOFT*, pages 371–374, 2005.

[19] G. Madl, S. Abdelwahed, and G. Karsai. Automatic Verification of Component-Based Real-Time CORBA Applications. In *Proceedings of RTSS*, pages 231–240, 2004.

[20] G. Madl, S. Abdelwahed, and D. C. Schmidt. Verifying Distributed Real-time Properties of Embedded Systems via Graph Transformations and Model Checking. *Real-Time Systems*, 33:77–100, 2006.

[21] G. Madl and N. Dutt. Tutorial for the Open-source Dream Tool. In *CECS Technical Report*, 2006.

[22] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. 1992.

[23] K. Richter, M. Jersak, and R. Ernst. A Formal Approach to MpSoC Performance Verification. *IEEE Computer*, 36:60–67, 2003.

[24] V. Subramonian, C. Gill, C. Sanchez, and H. Sipma. Reusable Models for Timing and Liveness Analysis of Middleware for Distributed Real-Time Embedded Systems. In *Proceedings of EMSOFT*, pages 252–261, 2006.

[25] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis - a case study. *Software Tools for Technology Transfer (STTT)*, 8(6):649–667, 2006.