Compositional Predictability Analysis of Mixed Critical Real Time Systems

Abdeldjalil Boudjadar[1], Juergen Dingel[2], Boris Madzar[2], Jin Hyun Kim[3]

[1] Linköping University Sweden
[2] Queen's University Canada
[3] INRIA Rennes France

Abstract. This paper introduces a compositional framework for analyzing the predictability of component-based embedded real-time systems. The framework utilizes automated analysis of tasks and communication architectures to provide insight on the schedulability and data flow. The communicating tasks are gathered within components, making the system architecture hierarchical. The system model is given by a set of Parameterized Stopwatch Automata modeling the behavior and dependency of tasks, while we use Uppaal to analyze the predictability. Thanks to the Uppaal language, our model-based framework allows expressive modeling of the behavior. Moreover, our reconfigurable framework is customizable and scalable due to the compositional analysis. The analysis time and cost benefits of our framework are discussed through an avionic case study.

1 Introduction

Since the Apollo Guidance Computer has been recognized as one of the first successful embedded systems designed early in the 60's, embedded software functions have been increasing in number, complexity and scale in the design of automotive and avionic systems. In some application areas, for example avionics, human life might be dependent on the reliability of such embedded systems which makes these systems highly critical. To demonstrate the reliability of safety critical systems, an intensive effort has been jointly undertaken by researchers and practitioners. Such a pursuit includes the definition of appropriate software engineering principles [23] (modularity, abstraction, separation of concerns, etc) and the development of powerful analysis tools [3, 27, 17].

A common execution requirement to be guaranteed when designing an embedded system is the response time [19], which is the end-to-end delay of the system execution. To be able to guarantee response times, 1) the execution times of actions must be bounded; 2) an analysis must demonstrate that the system produces its results under all relevant circumstances and all ways to resolve internal non-determinism (due to, e.g., concurrency and communication delays) and external non-determinism (due to, e.g., changes in input values/arrival times).

Predictability [16] has been identified as an input related requirement. It ascertains that the externally observable behavior of a process or a system remains the same despite internal non-determinism while removing external non-determinism (i.e., keeping the inputs and their timing unchanged).

Proving the predictability [25] means that the system analysis is successfully passed regarding both data flow and time-constrained behavior under any execution assumption, for example concerning failure and workload. An example of the predictability property is the Emergency Brake System [26] mounted in Volvo FH truck series since 2013 to avoid rear end collisions. Such a feature is a component of the Adaptive Cruise Control (ACC) system. Once the radar of a moving truck discovers an obstacle on the route of the truck, it communicates the distance information to a computation process that calculates the braking pressure to be applied based on the obstacle distance and the truck speed and delivers the braking pressure value to the braking system. The radar component is a composition of sensors and cameras. A danger state is determined by the presence of a stationary or a moving vehicle just in the front of the truck with a very slower speed than the truck's. The computation must output the correct brake pressure at the expected time, which is a couple of micro seconds after the detection of the obstacle. An unpredictable computation process might deliver different outputs in response to the same inputs, which could result in bugs that are hard to detect.

Different techniques have been introduced to analyze the predictability of real-time systems [15, 14, 12, 28], where the analysis does not leverage the system structure and systems are analyzed monolithically. This may lead to a state space explosion, making large systems non-analyzable. To the best of our knowledge, compositional analysis techniques for predictability have not received a lot of attention in the literature (discussed in Section 3).

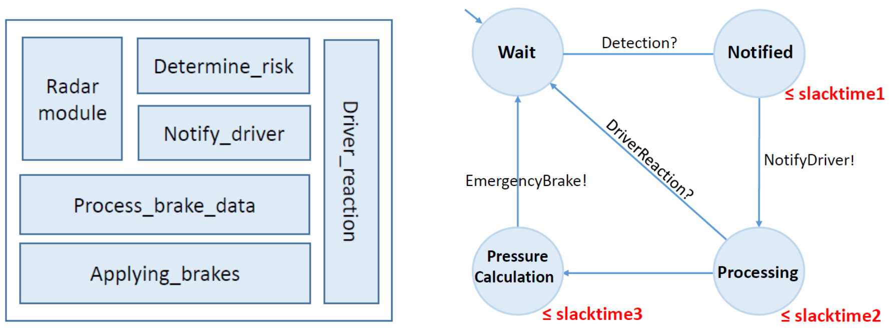
By compositional analysis [5], we mean that the analysis of a system relies on the individual analysis of its components separately, since they are independent. In such a design architecture, when a component violates its requirements it does not affect the execution of other components because the faulty component cannot request more than the resource budgeted by its interface (Section 5.2).

The system architecture we consider in this paper is structured in terms of components having different criticality levels. During execution, criticality levels will be used as static priorities to sort components. Each component is the composition of either other components (hierarchical) or basic processes (periodic tasks) having deterministic behavior. Each component will be analyzed individually and independently from the other system components thanks to its abstraction through an interface. We use parameterized stopwatch automata (PSA) to model the system while we use Uppaal toolsuite for simulation and formal analysis. The contributions of this paper include:

- How to support the predictability of hierarchical real-time systems through certain design restrictions.
- A scalable predictability analysis framework due to the component-based design and compositional analysis.
- Flexible and customizable framework due to the parametrization and instantiation mechanism of Uppaal.

The rest of the paper is organized as follows: Section 2 motivates the predictability analysis through an industrial example. Section 3 cites relevant related work. Section 4 introduces the predictability notion we adopt as well as schedulability as a sufficient condition for the predictability. In Section 5, we introduce a compositional analysis technique. Section 6 shows our model-based analysis for the predictability of component-based real-time systems using the Uppaal. Section 7 presents a case study. Section 8 concludes the paper.

2 Motivating Example



(a) (b)

Fig. 1: Volvo's Emergency Brake System.

Fig. 1 depicts the structure (1.(a)) and abstract behavior (1.(b)) of Volvo's emergency brake system mentioned above. The system consists of 6 concurrent components, each of which is given a set of timing attributes as well as a priority level. Once an input is generated by component `Radar module`, the component `Determine_risk` determines whether a potential obstacle is present or not. The component `Notify_driver` is responsible for notifying the driver in case a risk occurs. Based on the driver reaction, received and analyzed by component `Driver_reaction`, the system decides which action to take next. If the driver reaction is continuously missing for a certain duration, component `Process_brake_data` calculates the necessary brake pressure according to certain input data such as distance, truck speed and obstacle speed. Once the pressure value is handed over to component `Applying_brakes`, it brakes the truck.

Fig. 1.(b) depicts an abstract behavior of the overall emergency brake system. The system execution is initially in state `Wait` waiting to be triggered by the radar (external sensor) via a signal through channel *detection*. Once such a notification occurs, the system moves to state `Notified` waiting for the emergency data acquisition before notifying the truck driver. The data communication could be done via shared memory, bus, etc. The maximum waiting time for data acquisition must not exceed **slacktime1** time units. If the data is communicated late during the allowed interval [0, slacktime1], the remaining distance to the collision will not be the same, i.e., much shorter, as the truck is moving. After notifying the driver, the system moves to the state `Processing` and keeps calculating the remaining distance and time to the collision until either the driver reacts, and thus moves to the initial state, or reaches a critical time **slacktime2** by which it moves to state `PressureCalculation`. The slack time is calculated on the fly according to the distance, the truck speed, the elapsed time since detection and the obstacle speed. Once the brake pressure is calculated, the system activates the hardware through a signal on channel *EmegencyBrake* and moves to the initial state. The pressure calculation must be done within **slacktime3** time units. A safety property expected from this system is that it must deliver the right brake pressure at the expected time (bounded by the slack times). The later the notification arrives, the stronger the brake pressure has to be. In fact, the brake pressure delivered at time x, is different of that delivered at time $x+1$, and strongly dependent to the input values and the acquisition time of such inputs. Moreover, such a brake pressure must be predictable in a way that it is the same whenever the system is in the same configuration (data arrival time, elapsed time since the collision detection, the initial distance, the truck speed, etc). If the brake pressure is wrongly calculated (not sufficient) or delivered late, the truck will probably collide with the obstacle.

3 Related Work

In the literature, several model-based frameworks for the predictability analysis of real-time systems have been proposed [15, 14, 12, 28]. However, only few proposals consider the behavior of system processes (tasks) when analyzing predictability. Moreover, to the best of our knowledge it is very rare that the system predictability is analyzed in a compositional way.

The authors of [12] presented a model-based architectural approach for improving predictability of performance in embedded real-time systems. This approach is component-based and utilizes automated analysis of task and communication architectures. The authors generate a runtime executive that can be analyzed using the MetaH language and the underlying toolset. However the tasks considered are abstract units given via a set of timing requirements. Without considering the concrete behavior of system tasks, the analysis could be pessimistic and may lead to over-approximated results.

The authors of [22] defined a predictable execution model PREM for COTS (commercial-off-the-shelf) based embedded systems. The purpose of such a model is to control the use of each resource in the way that it does not exceed its saturation limit. Accordingly, each resource must be assigned at the expected time thus avoiding any delay at the operation points. This work focuses on resource utilization rather than data flow in case of communicating architectures. Moreover, analyzing the whole system at once might not be possible.

Garousi *et al* introduced a predictability analysis approach [15], for real-time systems, relying on the control flow analysis of the `UML 2.0` sequence diagrams as well as the consideration of the timing and distribution information. The analysis includes resource usage, load forecasting/balancing and dynamic dependencies. However, analyzing the whole system at once makes the identification of faulty processes/components not trivial.

The authors of [4] introduced a compositional analysis technique enabling predictable deployment of component-based real time systems running on heterogeneous multi-processor platforms. The system is a composition of software and hardware models according to a specific operational semantics. Such a framework is a simulation-based analysis, thus it cannot be used as a rigorous analysis means for critical systems.

Our paper introduces a compositional model-based framework for the predictability analysis of component-based real time systems, so that faulty components can easily be identified. The framework uses the expressive real-time formalism of parameterized stopwatch automata to describe the system/components behavior. We rely on the advances made in the area of model-checking by analyzing each component formally using the Uppaal model checker. The compositionality and parametrization lead our framework to be scalable and flexible.

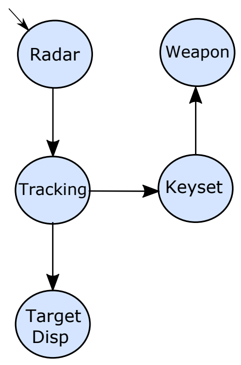
4 Predictable Real time Systems

Concurrent real-time systems [18] are usually specified by a set of communicating processes called tasks. Each task performs a specific job such as data acquisition, computation and data actuation. Moreover, tasks are constrained by a set of features, such as roundness and execution time, as well as a dependency relation capturing the data flow between processes.

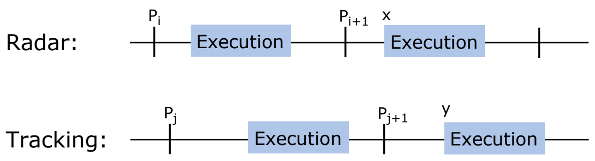
- *Roundness* includes the activation rhythm (periodic, aperiodic, sporadic) and the necessary time interval for each activation.
- *Execution time* specifies the amount of processing time required to achieve the execution of one task activation on a given platform.
- *Dependency* [10] describes the communication and synchronization order between tasks, meaning that a dependent task cannot progress if the task on which it depends has not reached a certain execution step or delivered a specific message.

Another property to be considered in case of dependency is the manipulation of correct data. So that when a task T_1 interacts with (or preempts) another task T_2, task T_1 must reload the data possibly modified by the execution of T_2 in order to avoid using out of date or inconsistent data. Powerful synchronization mechanisms enable to capture the interaction, and thus determine the time point at which the data produced by a task must be delivered to the consumer task.

In the literature, recent work [22, 2] enhances the predictability of real-time systems by restraining the observability of data in such a way that a consumer



(a) Dependency



(b) Restricted observability

Fig. 2: Example of dependency and restricted observability.

task can only access the data produced by a `run-until-completion` execution of the corresponding producer task. Moreover, such data must be produced before the consumer starts it current execution. For data consistency, tasks read and write data only on the beginning and the end of their period execution respectively. This implies that any data update made after the release of a given task will be ignored by that task for the current execution. This notion of run-to-completion data consistency is called *restricted observability* [2].

For example, if a consumer task synchronizes with a producer task, the legal data values to be used by the consumer after the synchronization must be the data issued before the consumer started its current job. This means that if the producer does not complete its execution before a synchronization point, the data value to be considered by the consumer for its current execution (potentially released at the synchronization time point) is not the value computed until the synchronization time but rather it is the data delivered at the termination of the previous execution period of the producer.

Fig. 2.(a) illustrates a dependency relation between different tasks of a mission control computer system. An arrow from one task T_1 to another task T_2 means that T_2 depends on T_1. Once the radar component captures the presence of a potential enemy engine it outputs data concerning the enemy position to the tracking task which in turn identifies the enemy status, speed, etc. Meanwhile, the tracking task unlocks the display task with the updated data for the target display on the screen. Once the enemy is positioned in a reachable distance, the keyset task will be unlocked to enable the aircraft pilot activating the weapon task to destroy the enemy engine.

Fig. 2.(b) depicts a data flow example following the restricted observability. For the period P_{j+1}, `Tracking` is released at time y while `Radar` is still running under its period P_{i+1}, the data to be considered by `Tracking` must be that issued by `Radar` before time x which means before the beginning of period P_{i+1}. Thus, the data considered by task `Tracking` during the period P_{j+1} is the update made by task `Radar` at the end of its execution for period P_i.

Technically, the predictability property we consider consists of 2 requirements: 1) data consistency; 2) execution order.

- *Data consistency* ensures that all tasks have the same observability of the data regardless of their dependencies. The non-preemption of tasks ensures that tasks access the shared data only at the scheduling time points, i.e a dependent task execution considers the data update made by the tasks on which it depends before its current release (scheduling) for the whole current period. Any other data update made externally during the task execution is ignored and can only be considered in the next scheduling of the task. A scheduling time point is the time instant when the execution of a running tasks is done and the scheduler releases another ready task. This approach to data observability is known as *predictable intervals* [22].
- *Execution order* between tasks follows the scheduling mechanism adopted by the real-time system, and must not be in contradiction with the dependency relation so that a dependent task cannot first execute before the tasks on which it depends.

Therefore, for real-time systems specified using non-preemptive tasks if the execution order, reflecting both scheduling mechanism and data consistency, is guaranteed then the schedulability is a sufficient condition for predictability [2]. Accordingly, predictability will simply be analyzed through schedulability.

Apart from the temporal partitioning [24] of the system workload to tasks, the separation of concerns [21] allows gathering collaborative and dependent tasks within components. Thus making the system architecture modular.

5 Compositional Framework for Predictability Analysis

In this section, we consider real-time systems structured as a set of independent components while we analyze system predictability, relying on the schedulability as a sufficient condition, in compositional way so that each component will be analyzed individually.

5.1 Hierarchical Real-Time Systems

Hierarchical scheduling systems [13, 11] have been introduced as a component-based representation of real-time systems, allowing temporal partitioning and separation of concerns. A major motivation of the separation of concerns [21] is that it allows isolation and modular design to accommodate changes in the system such that the impact of a change is isolated to the smallest component. An example of the increasing use of hierarchical scheduling systems is the standard ARINC-653 [1] for avionics real-time operating systems.

An example of a hierarchical scheduling system running on a single core platform is depicted in Fig. 3. It consists of 2 independent components, `Component1` and `Component2`, scheduled by the system level according to FPS (Fixed Priority Scheduling). For compositionality purposes, each component is given an

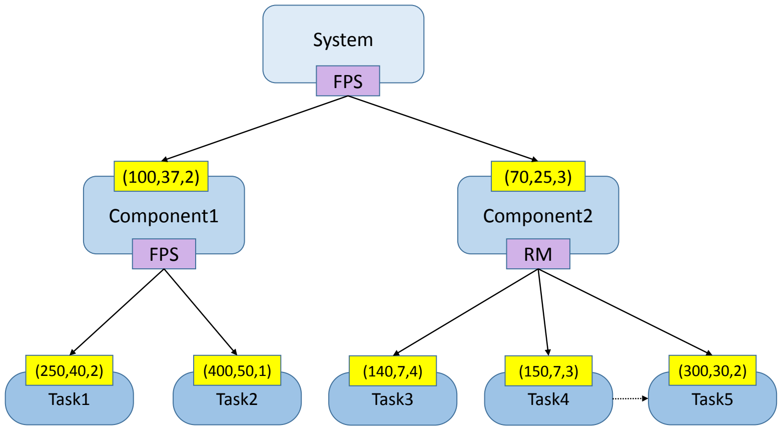


Fig. 3: Example of a hierarchical scheduling system.

interface (*period, budget, criticality*) e.g. (100,37,2) for `Component1`, where *budget* is the CPU time required by component for a time interval *period*. In our context, *criticality*[1] is handled as static priority to sort components at their parent level, so that in Fig. 3 `Component2` has priority over `Component1` ($2 < 3$). Each component in turn is a composition of tasks scheduled according to a local scheduler, FPS for `Component1` and RM (Rate Monotonic) for `Component2`. Each task is also assigned an interface *(period, exectime, prio)*, where *exectime* and *prio* are respectively the execution time and priority. Of course, the priority will be considered if a static priority scheduling scheduler is adopted. We also consider dependencies between tasks (the dashed arrow from `Task4` to `Task5`), so that the execution of a dependent task (`Task5`) cannot start until the task on which it depends (`Task4`) finishes it execution. In this work, we only consider periodic non-preemptive tasks.

At the system level, each component will be abstracted as a task given by the interface *(period, budget, criticality)* regardless of its child tasks. The interface of a component is a contract that the system level supplies such a component with *budget* CPU time every time interval of size *period*. Once a component is scheduled by the system level, it schedules one of its local tasks according to its scheduler, i.e. a component can trigger its child tasks only when it is allocated the CPU resource.

5.2 Compositional Analysis

By compositional analysis [5] we mean that the analysis process of a system relies on the individual analysis of each component separately, since components are independent. In such a design architecture, when a component violates its

[1] We do not consider the criticality related features like fault tolerance for soft critical components.

requirements it does not affect the execution of other components. The misbehavior cannot propagate because the faulty component, even though it is not satisfied with the resource budget it has been granted, cannot request more than the resource budgeted by its interface. Thus, the other concurrent components will not be deprived and remain supplied with the same budgeted resource amounts as in case of the successful behavior.

The analysis of each component consists in checking the feasibility of its tasks against its interface (*period, budget*), which is a guarantee that the component always supplies its tasks with the budgeted resource amount every period. To check that the tasks are feasible whatever the budget supply time, we consider all possible scenarios. We model the resource supply by a periodic process (*supplier*) having a non-deterministic behavior. For each period, the supplier provides the resource amount specified in the component interface (*budget*). Thereafter, we use a model checker to explore the state space, by considering all potential supply times, and verify whether all tasks are satisfied for all supply scenarios. For further description and illustration of our compositional analysis technique, we refer readers to [7, 8].

Depending on the interpretation of the deadline miss, the faulty component can either be suspended for the current period execution, discarded from the system (blocked) or just be kept running. The deadline miss interpretation strongly depends on the criticality and the application area of the failed component/system. Since we are considering criticality, in our framework the occurrence of a deadline miss implies a suspension of the execution, thus tasks termination (by deadline) is not guaranteed (the system is not schedulable). This implies that tasks cannot output data at the expected time (deadline), thus violate the predictability property.

5.3 Conceptual Design

Basically, the dependency relation can be viewed as order on the tasks execution in the way that a dependent task cannot run while the task on which it depends does hand out the event or data expected by the dependent task (in our context it is just a run-to-termination of the task execution for the current period). Tasks are usually given with a period *period*, an offset *offset*, an execution time *execTime*, a priority *prio* and a deadline *deadline*. Moreover, in our framework we consider a dependency relation *Dependency* between tasks. Throughout this paper we assume that the task period is greater or equal to the deadline. Moreover, the deadline must be greater than the execution time.

Fig. 4 depicts a conceptual model of tasks with dependency. The task is initially in state Wait Offset expiry waiting for the expiration of its offset. In state Wait dependency, the task waits execution termination of the immediate tasks on which it depends while its deadline is not missed yet. Once a task obtains the requested inputs it becomes ready to be scheduled and thus waiting for the CPU. A ready task moves to state Running when it is scheduled. Since the task behavior we consider is not preemptive, a scheduled task keeps running until satisfying the execution requirement or missing its deadline by which it joins the

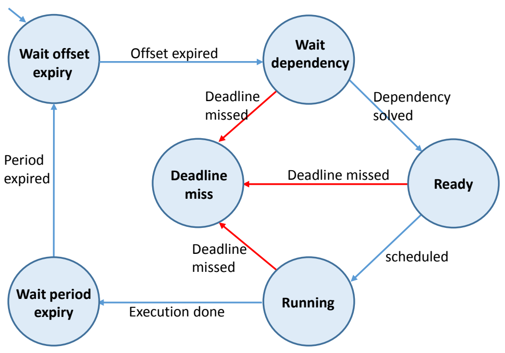


Fig. 4: Conceptual model of tasks.

state Deadline miss. After having satisfied the execution requirement the task enters state Wait period expiry waiting for the expiry of its current period.

Namely, the dependency relation is a direct acyclic graph where nodes represent tasks execution and transitions are the dependency order. A transition from a node to another means once the execution of the source node task is done the target node task is unlocked. Of course this does mean that the execution of such a task will start immediately but only becomes ready to be scheduled. A task must not depend to its dependent tasks nor to the tasks depending to one of its dependent tasks so far. The dependency of task must be applied for each period. Accordingly, a dependent task waits for its dependency to be satisfied whenever a new period starts. In turn, such a task unlocks its dependent tasks just for the execution of their current periods.

6 Uppaal System Model

Uppaal [3] is a tool environment for modeling, simulation and formal verification of real-time systems modeled as composition of inter-communicating processes. Each process is an instance of a template model. Our system model consists of a set of independent components, each of which is modeled separately and will be analyzed individually. Each template is a Parameterized Stopwatch Automata (PSA), offering the ability to use stopwatch clocks [9] and instantiation with different parameters.

Components Modeling. Each component is given by an interface (*period, budget, criticality*), a local scheduler and a workload. The workload of a component is either a set of tasks (i.e., basic component) or other components (i.e., hierarchical component). Components are independent and viewed by their parents as single periodic tasks having deadlines the same as periods. Such components are scheduled by their parent level's scheduler according to their *criticality*. Each component consists of a task model, a scheduler model, a CPU resource model, a supplier model [5] and a dependency relation.

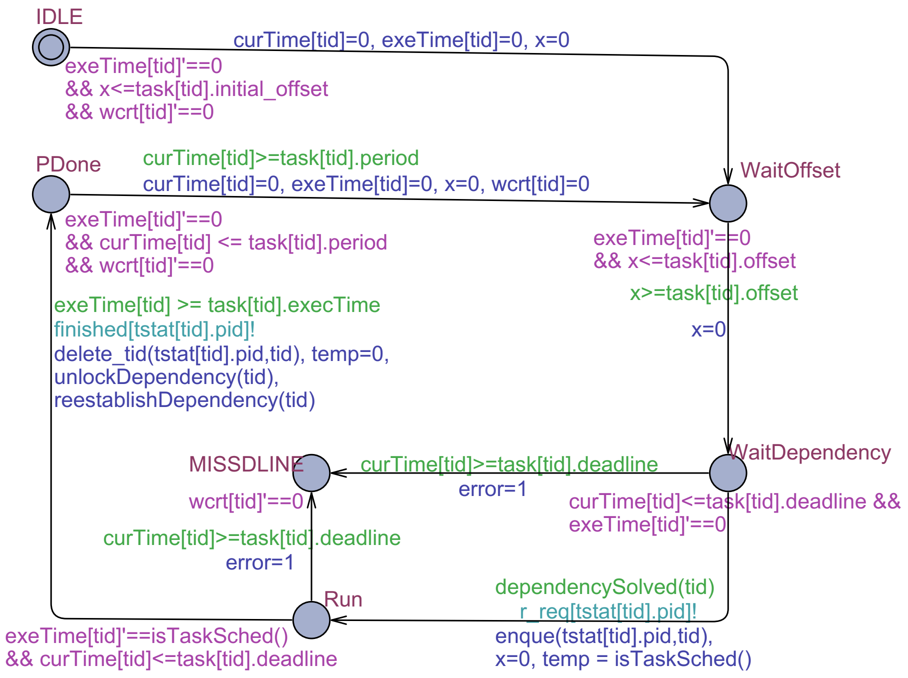

IDLE
curTime[tid]=0, exeTime[tid]=0, x=0
exeTime[tid]'==0
&& x<=task[tid].initial_offset
&& wcrt[tid]'==0
PDone
curTime[tid]>=task[tid].period
curTime[tid]=0, exeTime[tid]=0, x=0, wcrt[tid]=0
WaitOffset
exeTime[tid]'==0
&& curTime[tid] <= task[tid].period
&& wcrt[tid]'==0
exeTime[tid]'==0
&& x<=task[tid].offset
x>=task[tid].offset
x=0
exeTime[tid] >= task[tid].execTime
finished[tstat[tid].pid]!
delete_tid(tstat[tid].pid,tid), temp=0,
unlockDependency(tid),
reestablishDependency(tid)
MISSDLINE
curTime[tid]>=task[tid].deadline
error=1
WaitDependency
wcrt[tid]'==0
curTime[tid]<=task[tid].deadline &&
exeTime[tid]'==0
curTime[tid]>=task[tid].deadline
error=1
dependencySolved(tid)
r_req[tstat[tid].pid]!
Run
exeTime[tid]'==isTaskSched()
&& curTime[tid]<=task[tid].deadline
enque(tstat[tid].pid,tid),
x=0, temp = isTaskSched()

Fig. 5: Task model.

Task Model. Tasks are instances of the task template with the corresponding attributes (*tid*, *period*, *offset*, *exectime*, *deadline*, *prio*) as parameters. The task identifier *tid* is used to distinguish between tasks. Fig. 5 shows the PSA template we designed to model tasks. We use two stopwatch variables exeTime[tid] and curTime[tid] to keep track of the execution time and the current time respectively of a given task *tid*. Such variables are continuous but do not progress when their derivatives are set to 0.

Once started, the task model waits for the expiry of the initial offset at location IDLE. At location WaitOffset, the task waits until its periodic offset expires then moves to location WaitDependency. At both locations IDLE and WaitOffset the stopwatch exeTime[tid] does not progress because the task is not running yet. At location WaitDependency, the task is waiting until either its deadline is missed (curTime[tid]$\geq$ task[tid].deadline) or its dependency gets unlocked (dependencySolved(tid)). The stay at such a location is constrained by the invariant curTime[tid]$\leq$ task[tid].deadline, during which the stopwatch exeTime[tid] does not progress. Once the deadline is missed, the task moves to location MISSDLINE. Otherwise, the task is ready and it requests the CPU resource through an event r_req[tstat[tid].pid]! on channel r_req and moves to location Run. Through such an edge, the task enqueues its identifier tid into the queue of the resource model identified by *pid*. In fact, location Run corresponds to both ready and running status thanks to the stopwatch. Once the task gets scheduled through function isTaskSched() it keeps running while it is scheduled and its execution requirement is not fully satisfied. Thus, the stopwatch exeTime[tid] measuring the execution time increases continuously while isTaskSched() holds, i.e., exeTime[tid]'==isTaskSched().

For analysis performance, whenever a deadline is missed the faulty task updates the global variable error to one. Thus, the schedulability will be checked upon the content of this variable. When the execution requirement *execTime* is satisfied, exeTime[tid]$\geq$task[tid].execTime, the task moves to location PDone waiting for the expiry of the current period. Through such an edge, the task releases the CPU, unlocks the dependent tasks waiting for such a termination and reestablishes its original dependency for the next period.

CPU Resource Model. Fig. 6 depicts the CPU resource model. Once it starts, the CPU resource moves to location Idle, because the initial location (with double circles) is committed, and waits for a request from tasks through channel r_req[rid]. Through a resource request, the CPU model moves to location ReqSched and immediately calls the underlying scheduler. At location WaitSched, the CPU model is waiting for a notification from the scheduler through which the CPU will be assigned to a particular task at location Assign. Such a task will immediately be removed from the resource queue by the edge leading to the location InUse. As we consider non-preemptive execution only, if a task requests the CPU while it is assigned to another task such a request will be declined. However the requesting task will immediately be enqueued. Whenever the CPU resource is released by the current scheduled task, the resource model calls the scheduler to determine to which task it will be assigned if the the queue is not empty (location ReqSched). Otherwise, the resource model moves to location Idle waiting for task requests.

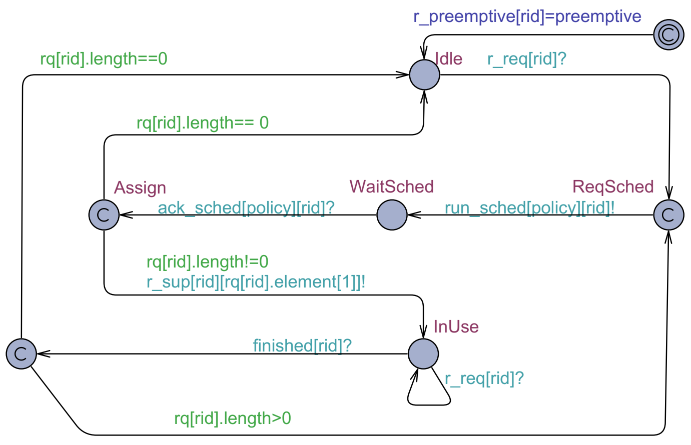


Fig. 6: CPU resource model.

Dependency Relation Modeling. Given n tasks, we model their dependencies by a matrix of 2 dimensions each of which has n elements. A row i represents the dependencies of all tasks to the task having identifier $tid = i$, whereas a column j states the identifiers of tasks on which the task $tid = j$ depends. The content of each cell is Boolean, so that $cell[i, j]$ states whether task $tid = j$ depends on the task having identifier $tid = i$. Accordingly, the dependencies of a task x are satisfied if the cells of column x are all *False*. Table. 1 shows the matrix representation of the dependency relation given in Fig. 2.(a).

Table 1: Implementation of the dependency relation of Fig. 2.(a).

Dependency	Radar	Tracking	Target	Keyset	Weapon
Radar	*False*	**True**	*False*	*False*	*False*
Tracking	*False*	*False*	**True**	**True**	*False*
Target	*False*	*False*	*False*	*False*	*False*
Keyset	*False*	*False*	*False*	*False*	**True**
Weapon	*False*	*False*	*False*	*False*	*False*

To manipulate the dependencies of tasks during components execution, we introduce the following functions:

– dependencySatisfied(tid) checks whether all tasks on which a given task *tid* depends have already updated their status to *Done* (execution finished for one period). This is done by verifying that all cells of the tid^{th} column of the dependency matrix are *False*.
– unlockDependent(tid) unlocks all tasks dependent on a given task *tid* when the execution of such a task is finished. This is done by updating the cells of row *tid* to *False*.
– reestablishDependency(tid) establishes the original dependency relation of a given task *tid* when its execution is done. This is done by updating the cells of column *tid*, corresponding to the tasks on which task *tid* originally depends, to *True*. Such a reestablishment is because, as stated earlier, the dependency relation is applicable every task period.

7 Case study

To show the applicability and scalability of our analysis framework, we modeled and analyzed an avionics system [20]. Table 2 lists the system components, tasks and their underlying timing attributes. Columns two and three list the criticality level and tasks of each component. Columns four to seven list the timing attributes of tasks, whereas the last column describes the tasks on which each task depends. Due to space limitation, we do not consider inter-component dependencies however it can simply be applied since our analysis is recursive where components are viewed by their parent levels as single tasks.

Table 2: Avionics Mission Control System

Component	Criticality	Tasks	p_i	e_i	d_i	$prio_i$	Task dependency
Display	1	Status Update (T_1)	200	3	200	12	T_2, T_3, T_5
		Keyset (T_2)	200	1	200	16	—
		Hook Update (T_3)	80	2	80	36	—
		Graph Display (T_4)	80	9	80	40	T_1, T_3
		Store Updates (T_5)	200	1	200	20	T_2
RWR	3	Contact Mgmt (T_6)	25	5	25	72	—
Radar	3	Target Update (T_7)	50	5	50	60	T_8
		Tracking Filter (T_8)	25	2	25	84	—
NAV	2	Nav Update (T_9)	59	8	59	56	T_{10}
		Steering Cmds (T_{10})	200	3	200	24	—
		Nav Status (T_{11})	1000	1	1000	4	T_9
Tracking	1	Target Update (T_{12})	100	5	100	32	—
Weapon	4	Weapon Protocol (T_{13})	200	1	200	28	T_{15}
		Weapon Release (T_{14})	200	3	200	98	T_{13}
		Weapon Aim (T_{15})	50	3	50	64	—
BIT	0	Equ Stat Update (T_{16})	1000	1	1000	8	—
Data Bus	2	Poll Bus (T_{17})	40	1	40	68	—

We consider that components having criticality levels less than 2 are not hard critical. Moreover, for the components having one task only, the component period, respectively budget, is the same as the child task period, respectively execution time. Since tasks are non preemptible and satisfy the restricted observability, we check predictability through schedulability. Table. 3 summarizes the analysis results. First, we calculate the minimum budget of each composite component using a binary checking while varying the component budget [6].

Table 3: Analysis results of the case study.

Component	Period	Budget	CPU utilization	Analysis time (s)	Memory space (KB)
Display	80	13	13/80	0.016	8852
Radar	10	2	2/10	0.016	7656
NAV	20	3	3/20	0.016	7784
Weapon	50	4	4/50	0.015	7748

The analysis time (15 and 16 milliseconds) is very low compared to the system size, while the used memory space is relatively acceptable. In a previous work [20], Locke *et al* estimated the resource utilization of the whole system to 85% without considering data flow time. In our paper, while considering data flow between certain tasks we estimated the resource utilization to 86.25%. Such a utilization is very high and leads the avionic system to be non-schedulable, in particular if the overhead time is also considered. Accordingly, the individual tasks cannot guarantee to output data before their deadlines, thus making the system unpredictable.

8 Conclusion

In this paper we have introduced a compositional model-based framework for the predictability analysis of real-time systems. The architecture we considered is hierarchical where components running on a single core platform may have different criticality levels. The system tasks are periodic and may depend on each other. We analyze each component individually by providing insight on the schedulability and data flow.

Our framework is set using Uppaal while the real-time formalism we used to model tasks and data flow is the stopwatch automata. We believe that our framework is scalable as long as the system is designed in terms of independent (average size) components.

A future work is the introduction of a new task model to capture data flow and analyze the predictability without considering the schedulability as a sufficient condition.

References

1. ARINC 653. Website. https://www.arinc.com/cf/store/documentlist.cfm.
2. C. Aussagues, D. Chabrol, V. David, D. Roux, N. Willey, A. Tournadre, , and M. Graniou. PharOS, a multicore OS ready for safety-related automotive systems: results and future prospects. In *ERTS2'10*, May 2010.
3. G. Behrmann, A. David, and K. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *LNCS*, pages 200–236. Springer Berlin Heidelberg, 2004.
4. E. Bondarev, M. Chaudron, and P. de With. compositional performance analysis of component-based systems on heterogeneous multiprocessor platforms. In *SEAA '06.*, pages 81–91, Aug 2006.
5. A. Boudjadar, A. David, J. Kim, K. Larsen, M. Mikuionis, U. Nyman, and A. Skou. hierarchical scheduling framework based on compositional analysis using Uppaal. In *FACS'13*, LNCS, pages 61–78. Springer, 2013.
6. A. Boudjadar, A. David, J. H. Kim, K. G. Larsen, M. Mikucionis, U. Nyman, and A. Skou. widening the schedulability of hierarchical scheduling systems. In *Proceedings of FACS 2014*, pages 209–227, 2014.
7. A. Boudjadar, A. David, J. H. Kim, K. G. Larsen, M. Mikucionis, U. Nyman, and A. Skou. A reconfigurable framework for compositional schedulability and power analysis of hierarchical scheduling systems with frequency scaling. *Science of Computer Programming-Journal*, In Press(X):25, 2015.
8. A. Boudjadar, J. H. Kim, K. G. Larsen, and U. Nyman. compositional schedulability analysis of an avionics system using Uppaal. In *Proceedings of the International Conference on Advanced Aspects of Software Engineering ICAASE*, pages 140–147, 2014.

9. F. Cassez and K. G. Larsen. the impressive power of stopwatches. In C. Palamidessi, editor, *CONCUR*, volume 1877 of *LNCS*, pages 138–152, 2000.
10. A. David, K. G. Larsen, A. Legay, and M. Mikucionis. schedulability of Herschel-Planck revisited using statistical model checking. In *ISoLA (2)*, volume 7610 of *LNCS*, pages 293–307. Springer, 2012.
11. Z. Deng and J. W.-S. Liu. scheduling real-time applications in an open environment. In *RTSS*, pages 308–319, 1997.
12. P. Feiler, B. Lewis, and S. Vestal. Improving predictability in embedded real-time systems. Technical Report CMU/SEI-2000-SR-011, Carnegie Mellon University, December 2000.
13. X. A. Feng and A. K. Mok. a model of hierarchical real-time virtual resources. In *RTSS '02*, pages 26–35. IEEE Computer Society, 2002.
14. J. Fredriksson. *Improving Predictability and Resource Utilization in Component-Based Embedded Real-Time Systems.* PhD thesis, Mälardalen University, 2008.
15. V. Garousi, L. C. Briand, and Y. Labiche. a unified approach for predictability analysis of real-time systems using UML-based control flow information. In *MoDELS*, volume LNCS 3844, 2005.
16. T. A. Henzinger. Two challenges in embedded systems design: predictability and robustness. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 366(1881):3727–3736, 2008.
17. G. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, May 1997.
18. J. Hooman. *Specification and Compositional Verification of Real-Time Systems (Book)*, volume 558 of *LNCS*. Springer Verlag, 1991.
19. M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.
20. C. Locke, D. Vogel, and T. Mesler. building a predictable avionics platform in Ada: a case study. In *Proceedings of RTSS*, pages 181–189, 1991.
21. M. Panunzio and T. Vardanega. A component-based process with separation of concerns for the development of embedded real-time software systems. *Journal of Systems and Software*, 96(0):105 – 121, 2014.
22. R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. a predictable execution model for COTS-based embedded systems. In *RTAS'11*, pages 269–279, April 2011.
23. S. L. Pfleeger and J. M. Atlee. *Software Engineering — Theory and Practice (4th Edition).* Pearson Education, 2009.
24. K. Purna and D. Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *Computers, IEEE Transactions on*, 48(6):579–590, Jun 1999.
25. J. Stankovic and K. Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, 1990.
26. Volvo Trucks Great Britain and Ireland. Driver support systems: Keeping an extra eye on the road. `http://www.volvotrucks.com/trucks/uk-market/en-gb/trucks/volvo-fh-series/key-features/Pages/driver-support-systems.aspx`.
27. F. Wang. Efficient verification of timed automata with BDD-Like Data-Structures. In *VMCAI'03*, volume 2575 of *LNCS*, pages 189–205. Springer Berlin, 2003.
28. S. Yau and X. Zhou. schedulability in model-based software development for distributed real-time systems. In *Proceedings of WORDS'02*, pages 45–52, 2002.