Modeling and Analysis of CPU Usage in Safety-Critical Embedded Systems to Support Stress Testing

Shiva Nejati[1], Stefano Di Alesio[1,2], Mehrdad Sabetzadeh[1], and Lionel Briand[1,2]

[1]SnT Center,
University of Luxembourg, Luxembourg

[2]Certus Software V&V Center,
Simula Research Laboratory, Norway

{shiva.nejati,stefano.dialesio,mehrdad.sabetzadeh,lionel.briand}@uni.lu

Abstract. Software safety certification needs to address non-functional constraints with safety implications, e.g., deadlines, throughput, and CPU and memory usage. In this paper, we focus on CPU usage constraints and provide a framework to support the derivation of test cases that maximize the chances of violating CPU usage requirements. We develop a conceptual model specifying the generic abstractions required for analyzing CPU usage and provide a mapping between these abstractions and UML/MARTE. Using this model, we formulate CPU usage analysis as a constraint optimization problem and provide an implementation of our approach in a state-of-the-art optimization tool. We report an application of our approach to a case study from the maritime and energy domain. Through this case study, we argue that our approach (1) can be applied with a practically reasonable overhead in an industrial setting, and (2) is effective for identifying test cases that maximize CPU usage.

1 Introduction

Many safety-critical systems, e.g., those in the avionics, railways, and maritime and energy domains, are increasingly relying on embedded software for control and monitoring of their operations. The safety-related software components of these systems are often subject to software safety certification, whose goal is to provide an assurance that the components are deemed safe for operation. Software safety certification needs to take into account various non-functional constraints that govern how software should react to its environment, and how it should execute on a particular physical platform [1]. These constraints, among others, include deadlines, throughput, jitter, and resource utilization such as CPU and memory usage [2, 3]. Reasoning about these constraints is becoming more complex in large part due to the multi-threaded design of embedded software, the shift towards multi-core and decentralized architectures for execution platforms, and the increasing complexity of real-time operating systems.

In this paper, we concentrate on a particular type of non-functional constraints, namely *CPU usage*, and provide a framework to derive test cases to verify that the CPU time used by a set of concurrent threads running on a multi-core CPU does not exceed a given limit, even under the worst possible circumstances. Keeping CPU usage low in safety-critical applications is not merely for general quality reasons, but rather an important safety precaution since with CPU usage above a certain threshold, the system may fail to respond in a timely manner to safety-critical alarms. For example, if a fire and gas monitor is starved of CPU time due to CPU overload, it can have a delayed or miss response to a fire or gas leak with potentially serious consequences. As a result, test cases that can stress the system to maximize CPU usage are crucial for certification

of safety-critical systems. Our approach to CPU usage modeling and analysis is driven by two main considerations:

1) Explicit modeling of time. Reasoning about CPU usage requires an explicit notion of time. Logic-based languages used for reasoning about concurrent software, e.g., most temporal logics, do not capture time explicitly [4]. Hence, while they can be used to reason about relative orderings of concurrent tasks, they cannot be used for computing constraints involving actual time values. We instead follow the common practice in standard languages such as the UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [5], where time should be explicitly expressed.

2) Search-based optimization. Our goal is to find testing scenarios that maximize CPU usage by a set of parallel threads running on a multi-core platform. We refer to this testing activity as *stress testing* [6]. We characterize the stress test scenarios, i.e., test cases, by *environment-dependent parameters* of the embedded software, e.g., the size of time-delays used in software to synchronize with hardware devices or to receive feedback from the hardware devices. To stress test the system, we choose the environment parameters in such a way that the system is pushed to use the maximum amount of CPU. Finding such stress test cases requires to search the possible ways that a set of real-time tasks can be executed according to the scheduling policy of their underlying real-time operating system. In our approach, the search for stress test cases is formalized using a constraint optimization model that includes (1) a set of constraints describing a declarative representation of the tasks, their timing constraints and priorities, and the platform-specific information, and (2) a cost function that estimates CPU usage. Our approach for deriving test cases, while it may not result in provable system safety arguments, can always provide test cases within a time budget and given a (potentially partial) set of declarative constraints characterizing the embedded software and its environment.

Contributions of This Paper. We develop an automated tool-supported solution for deriving test cases exercising the CPU usage requirements of a set of embedded parallel threads running on a *multi-core* CPU. Specifically, we make the following contributions:

- A conceptual model that captures, independently from any modeling language, the abstractions required for analyzing CPU usage requirements in embedded systems (Section 3.1). To simplify the application of our conceptual model in standard Model-Driven Engineering (MDE) tools, we provide a mapping between our conceptual model and UML/MARTE (Section 3.2).
- Casting of the CPU usage problem as a constraint optimization problem over our conceptual model (Section 4). If done effectively, this enables the use of mature constraint optimization technologies that have not been used so far to address this type of problems. We provide an implementation of our approach in the COMET tool [7] (Section 4). COMET comes with efficient implementations of various search algorithms and is widely used in Operations Research for solving optimization problems.
- An industrial case study from the maritime and energy domain concerning safety-critical IO drivers. IO drivers are some of the most complex software components in the maritime and energy domain with sophisticated concurrent designs and many real-time properties (Sections 2). Our case study shows that our approach (1) can be applied with a practically reasonable overhead, and (2) can identify test cases maximizing CPU usage within time constraints (Section 5).

Structure of the Paper. In Section 2, we motivate our work using a specific industrial context. In Section 3, we present our conceptual model and show how it can be mapped to UML/MARTE. In Section 4, we formulate CPU usage as a constraint optimization problem. We provide an evaluation of our approach in Section 5. We compare with related work in Section 6, and conclude the paper in Section 7.

2 Motivating Case Study

We motivate our work with a class of safety-critical I/O drivers from the maritime and energy industry. These drivers are used in fire and gas monitoring applications, and their overall objective is to transfer data between control modules, and hardware devices, i.e., detectors. The variations between different drivers are mainly due to the different communication protocols that they implement in order to connect to different types of detectors built by different vendors. Such drivers are common in many industry sectors relying on embedded systems.

One of the main complexity factors in drivers is that they need to bridge the timing discrepancies between hardware devices and software controller modules. Hence, their design typically consists of several parallel threads communicating in an asynchronous manner to enable smooth data transfer between hardware and software. Often, several execution time constraints are included in the drivers' requirements to ensure that the flexibility in the design of the drivers does not come at the cost of overusing the resources of the execution platform. An example of such constraints is the following: *An I/O driver shall, under normal conditions, not impact heavily on the CPU time. When only one driver instance is running, the idle CPU time shall be above 80%.*

There are three important context factors from the case study influencing our formulation of the CPU usage problem in this paper:

1. Different instances of a given driver are independent in the sense that they do not communicate with one another and do not share memory.
2. The purpose of the CPU usage constraints is to enable engineers to estimate the number of driver instances of a given monitoring application that can be deployed on a CPU. These constraints express bounds on the amount of CPU time required by *one* driver instance. Our analysis in this paper, therefore, focuses on individual driver instances. The independence of the drivers (first factor above) is key to being able to localize CPU usage analysis to individual instances in a sound manner.
3. The drivers are not *memory-bound*, i.e., the CPU time is not largely affected by the low-bound memory allocation activities such as transferring data in and out of the disk and garbage collection. To ensure this, the partner company (over-) approximates the maximum memory required for each driver instance by multiplying the number of detectors connected to the driver instance and the maximum size of data sent by each detector. Execution profiles at the partner company indicate that the drivers are extremely unlikely to exceed this limit during their lifetime.

Figure 1(a) illustrates an activity diagram capturing the overall architecture of the I/O drivers we focus on. Each driver consists of three parallel threads: Two of these threads, **pullData** and **pushCmd**, are executed periodically upon receipt of a trigger, i.e., **scan**. The **IODispatch** thread, however, is enclosed within an infinite (unconditional) loop. The **pullData** thread receives (pulls) data from sensors/human operators/control modules, then puts the data in an appropriate command form, and finally

sends it to the **IODispatch** thread through a shared memory storage. The **pushCmd** thread receives commands from the **IODispatch** thread via another memory storage, and transfers them to the Fire Monitoring Systems (FMS). The memory storages in Figure 1 are shared only between the threads of one driver instance.

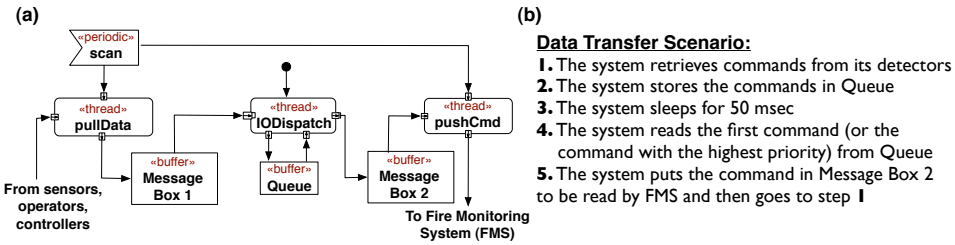


Fig. 1. Driver case study: (a) Overview of the drivers' architecture. (b) The CPU intensive scenario of drivers. This scenario is subject to stress testing regarding CPU usage.

Drivers in our partner company have four modes of operation: maintenance, normal, initial and termination. Only the normal mode is critical with regard to CPU usage. In this mode, the connections with FMSs are established, and the *data transfer* scenario is enabled. The data transfer scenario of the drivers for an example communication protocol is shown in Figure 1(b). It describes a uni-directional communication where some delay is injected between the commands in each transmission iteration to ensure that the commands are received by FMSs at a slow enough rate so that the FMS can process them. To show that drivers satisfy their CPU usage requirement, we focus on data transfer scenarios of drivers only because other driver scenarios are not CPU intensive

The drivers run on a CPU with three separate cores. The operating system used is VxWorks [8] – a *Real-Time Operating System (RTOS)*. RTOSs share many features with general-purpose operating systems, but in addition have specialized kernels and a process scheduler that takes into account real-time constraints [3]. The installation of VxWorks in our study uses a *fixed priority preemptive scheduler*. This scheduling policy does not allow for a lower-priority task to execute while a high-priority task is ready. The **pullData**, **pushCmd**, and **IODispatch** threads communicate in an asynchronous way through buffers implemented using message queuing utilities provided by VxWorks. In the driver implementation, all accesses to the shared buffers are properly protected by semaphores and can potentially be blocking.

To estimate the CPU time used by a driver, we need to first include in our design models timing information such as how long it takes for the threads to run and their frequency. We then need to specify how the CPU usage can be characterized based on the input timing information. This requires capturing the *concurrent* dependencies between the threads, how these threads *communicate*, how the RTOS scheduler *preempt*s the threads, and how the threads can run on a *multi-core processor*. In the subsequent sections, we provide our solution that can address all these details.

3 Modeling Guidelines

In this section, we first provide a *conceptual model* that captures the timing abstractions necessary for analyzing CPU usage (Section 3.1). We then show how the abstractions are mapped to the UML/MARTE metamodel (Section 3.2).

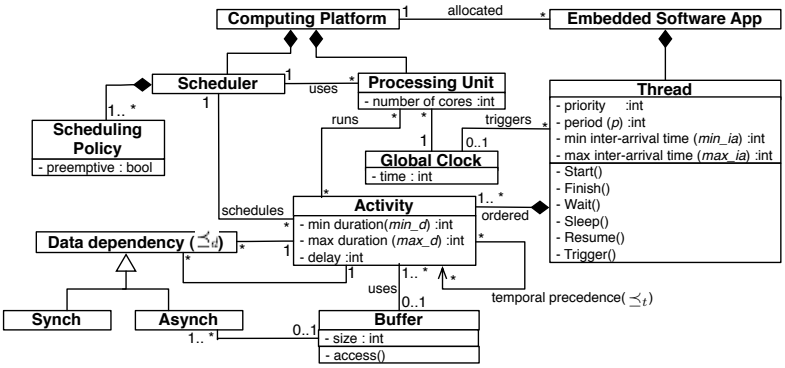


Fig. 2. The conceptual model characterizing the information required for CPU usage analysis.

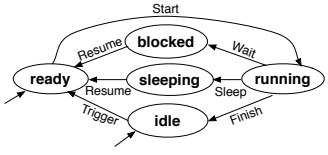
3.1 Conceptual Model

The conceptual model depicted in Figure 2 and explained below specifies the information required for analysis of CPU usage:

Thread. An embedded software application consists of a set $J = \{j_1, \ldots, j_n\}$ of parallel threads. A thread $j \in J$ can be *periodic* or *aperiodic*. Periodic threads, which are triggered by timed events, are invoked at regular intervals and as such their execution time is bounded by the (fixed) length of one interval, denoted $p(j)$ [9]. Any thread that is not periodic is called aperiodic. Aperiodic threads have irregular arrival times. In general, there is no limit on the execution time of an aperiodic thread, but one can optionally have a minimum inter-arrival time $min\_ia(j)$ and a maximum inter-arrival time $max\_ia(j)$ indicating the minimum and maximum time intervals between two consecutive arrivals of the event triggering the thread, respectively [10].

A common use of periodic threads is when we need to send/receive data regularly (e.g., **pullData** and **pushCmd** in Figure 1). In contrast, aperiodic threads are often used to process asynchronous events/communications (e.g., **IODispatch** in Figure 1). Each periodic or aperiodic thread has a priority, denoted $priority(j)$ that is used by a priority-based scheduler to determine which thread should be running at each time.

During its lifetime, a thread may perform the following lifecycle operations: (1) Start(): to start execution after having been assigned to a CPU core by the scheduler. (2) Finish(): to complete its execution. (3) Wait(): to wait in order to synchronize with another thread or to acquire some resource. (4) Sleep(): to go to sleep. (5) Resume(): to indicate to the scheduler that it is ready to resume execution after a previous block or sleep period. (6) Trigger(): to indicate to the scheduler that it is ready to start a new execution in response to a new triggering event after having completed a prior round of execution. The above state machine shows the lifecycle of a typical thread. A thread consumes CPU time only when it is running.



Activity. An *activity* is a sequence of operations in a thread that can execute without needing to release the CPU until its very last operation. The only situation where an activity releases the CPU is when it is preempted by a (preemptive) scheduler so that

the CPU can go to another activity belonging to a thread that has a higher priority. In other words, an activity is a sequence of operations that has Wait(), Sleep() or Finish() as its last operation but nowhere else in the sequence. Each thread $j \in J$ has a sequence $(a_1, \ldots, a_{m_j})$ of activities. We denote the set of all activities within a software application by A. Each activity a has an *estimated* minimum and maximum execution time denoted by $min\_d(a)$ and $max\_d(a)$, respectively. For each activity a, we denote the thread that owns that activity by $thread(a)$. Each activity a has a priority, inherited from its owning thread, i.e., $priority(thread(a))$. When activities end with a Sleep() operation, they are followed by a period of sleeping time. We use $delay(a)$ to denote the duration of the sleeping time. Activities of parallel threads can be related to one another by two kinds of relations: temporal precedence and data dependency.

Temporal precedence. When an activity a must be executed prior to an activity a', i.e., a is a prerequisite of a', we say that a (temporally) precedes a' and denote this by $a \preceq_t a'$ where $\preceq_t \subseteq A \times A$. Temporal precedence relates activities belonging to the same thread only.

We unroll the loops by copying the loop body a certain number of times. The number of unrollings can be chosen as an input parameter, and depends on the amount of time during which we choose to observe execution of the threads in our case study (see the notion of observation time interval discussed in Section 4). Temporal precedence then indicates that the activities in the first copy of the loop body precede those in the second copy, and those in the second copy precede those in the third copy, and so on – an example is given in Section 3.2. We also ensure that the last activity in the ith copy of the loop is followed by the first activity in the $(i+1)$st copy of the loop (see loop constraints in Section 4).

Data dependency. An activity a may depend on another activity a' because a requires some data that is computed by a'. We denote this relation by $a' \preceq_d a$ where $\preceq_d \subseteq A \times A$. For any data-dependent pair of activities, we need to specify whether the communication is synchronous or asynchronous. The activities related by a data dependency relation may or may not belong to the same thread.

Buffer. Asynchronous communications may or may not use buffers. Buffer accesses by activities are protected by semaphores, and are blocking. Hence, each buffer access within an activity implies an implicit Wait() operation and indicates the last operation of that activity. Therefore, each activity can be related to at most one buffer. Also, at most one activity can access a given buffer at any point in time. The time an activity is blocked, waiting for a shared buffer, is determined by our scheduling constraints and is zero when the buffer is not locked by any other activity (see Section 4).

Computing Platform and Global Clock. In addition to information about the software application itself, we need information about the characteristics of the underlying computing platform. In particular, we require knowledge of the number of CPU cores, denoted c, which indicates the maximum number of parallel activities that the CPU can host. We further need to know whether the scheduling policy used by the real-time scheduler is preemptive or non-preemptive. Lastly, we need a (real-time) clock to model time-based events/triggers.

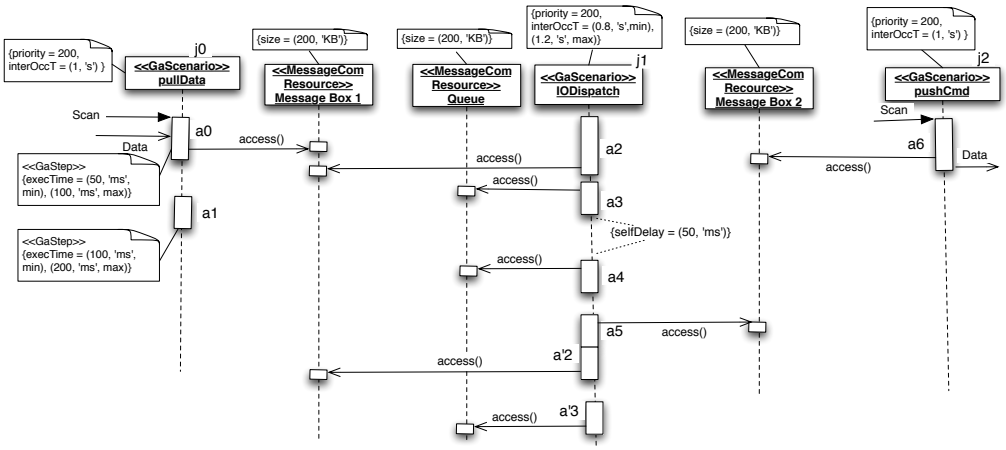


Fig. 3. A sequence diagram capturing the data transfer scenario in Figure 1(b). The diagram is augmented with MARTE timing and concurrency stereotypes and attributes.

3.2 Mapping to UML/MARTE

In this section, we demonstrate how the abstractions in Figure 2 are mapped to the UML/MARTE metamodel. This mapping shows the feasibility of extracting the abstractions required for CPU usage analysis from standard modeling languages supported by industry strength tools. We begin by first describing the abstractions that are already present in UML. We then show how missing timing and concurrency aspects can be mapped to MARTE.

In UML, Active Objects have their own threads of control, and can be regarded as concurrent threads [11]. Active objects in UML sequence diagrams can be associated to lifelines with several Execution Specifications that match activities in our conceptual model in Figure 2. The notation for describing synch and asynch communication already exists in UML sequence diagrams where an Occurrence Specification indicates a sending or a receiving of a message.

Figure 3 shows a sequence diagram capturing the data transfer scenario described in Figure 1(b). As shown in the figure, each thread in the driver application has an object with lifeline. Similar to threads and as shown in Figure 3, buffers correspond to passive objects in UML, and can be represented using lifelines as well.

We represent each activity of a thread using an *activation* or an *execution specification* (a thin box on the lifeline of a thread that shows the interval of time that the thread is active). Each thread lifeline is made up of a sequence of activations corresponding to its activities: thread j_0 is composed of activities a_0 and a_1; thread j_1 is composed of a_2, a_3, a_4, a_5, a'_2, and a'_3; and thread j_2 is composed of a single activity, a_6. Activities a'_2 and a'_3 are repetitions of a_2 and a_3, respectively. Due to space reasons, we have not shown any repetition of a_4 or a_5, or any further repetitions of a_2 or a_3. As mentioned earlier, we use constraints to ensure that a_5 is followed by a'_2 (see Section 4).

The order of activations on a thread lifeline implies the temporal precedence between activities of that thread. For every pair a, a' of activities, $a \preceq_t a'$ if a and a' belong to the same thread j and a precedes a' as indicated by the lifeline of j. For example, in Figure 3, we have $a_0 \preceq_t a_1$ and $a_2 \preceq_t a_3 \preceq_t a_4 \preceq_t a_5 \preceq_t a'_2 \preceq_t a'_3$.

In sequence diagrams, a synchronous message from an activity a to an activity a' is shown using a solid arrow with a full head; an asynchronous message is shown by a solid arrow with a sticky head. Synchronous communication is blocking and does not require a buffer by default. I.e., the sending activity must wait until the receiving activity is ready to receive messages. Asynchronous communications may or may not use buffers. In our case study, and hence in the sequence diagram of Figure 3, all communications are asynchronous and buffered. Based on the information obtained from the drivers' design, for the activities in Figure 3, we have $a_0 \preceq_d a_2$, $a_3 \preceq_d a_4$, and $a_4 \preceq_d a_6$.

Even though UML sequence diagrams can already capture several concepts in the Embedded Software Application package in Figure 2, the schedulability concepts, and the timing and concurrency attributes in that figure do not have appropriate counterparts in UML. These concepts are captured by extensions of UML, in particular MARTE, which is geared towards both the real-time and embedded system domains.

MARTE provides a Generic Quantitative Analysis Modeling (GQAM) sub-profile intended to provide a generic framework for collecting information required for performance and schedulability analysis. The domain model of this sub-profile includes two key abstractions that closely resemble our notions of thread and activity respectively: *Scenario* and *Step*. Step is a unit of execution, and Scenario is a sequence of steps. We map $\langle\langle$GaStep$\rangle\rangle$ (resp. $\langle\langle$GaScenario$\rangle\rangle$) which is a stereotype representing the notion of Step (resp. Scenario) in the domain model of GQAM to our notion of activity (resp. thread). These two stereotypes can be applied to a wide set of behaviour-related elements in UML 2.0 metamodel, and in particular, to UML sequence diagrams. We also map our notion of buffer to $\langle\langle$MessageComResource$\rangle\rangle$ which represents artifacts for communicating messages among concurrent resources.

MARTE includes a list of measures that are widely-used for analysis of real-time properties of embedded systems. The majority of these are applied to Steps and Scenarios, creating their sets of quantitative attributes. The top two rows of Table 1 show the mapping between our timing attributes to those of $\langle\langle$GaScenario$\rangle\rangle$ and $\langle\langle$GaStep$\rangle\rangle$ in MARTE. For example, we map interOccTime, the time interval between two successive occurrences of scenarios, to period or (max/min) inter-arrival times of threads, and execTime, the execution time of a step, to (min/max) duration of activities. Note that both of these measures can be specified either as single values or as max/min intervals. As an example, the sequence diagram in Figure 3 is augmented with the timing and concurrency stereotypes and attributes from MARTE.

We identified only one discrepancy in our mapping: In MARTE, individual steps have a *priority* attribute, indicating the priority of the step on their processing host, but this priority attribute does not directly apply to scenarios. At the implementation level, however, it is common to define priorities for threads rather than for steps within the threads. Hence, we specified priorities at the level of threads (Scenarios) and not for individual activities (Steps). In our mapping, we assume that the steps within a scenario have the same priority that carries over to the scenario which is a composite entity.

Information about the computing platform in Figure 2 is not captured on the sequence diagram but can be represented using MARTE streotypes applied to UML class diagrams. The GRM::Scheduling sub-profile already includes the schedulability concepts of Figure 2, i.e., $\langle\langle$Scheduler$\rangle\rangle$ and $\langle\langle$SchedulingPolicy$\rangle\rangle$. Finally, we map process-

ing units in Figure 2 to ⟨⟨ComputingResource⟩⟩ from GRM::ResourceType sub-profile, and the global clock to ⟨⟨LogicalClock⟩⟩ from TimeAccesses::Clocks sub-profile. The latter allows us to define regular triggers/events in RTOSs, e.g., scan in Figure 1.

Table 1. Mapping abstractions in Figure 2 to UML/MARTE.

	Concept	MARTE StereoType/attributes	MARTE Sub-Profile
Embedded Soft. App.	Thread - priority - period, - (min/max) inter-arrival time	«GaScenario» - *priority: NFP_Integer - interOccT: NFP_Duration[*] «TimedConstraint»	GQAM:: GQAM_Workload TimedConstraints
	Activity - (min/max) duration - delay	«GaStep» - execTime: NFP_Duration[*] - selfDelay: NFP_Duration[*]	GQAM:: GQAM_Workload
	Buffer - size - access()	«MessageComResource» - messageSizeElements: ModelElement [0..*] - sendServices/receiveServices: BehavioralFeature [0..*]	SRM:: SW_Interaction
Comp. Plat.	Scheduler	«Scheduler»	GRM::Scheduling
	Scheduling Policy	«SchedulingPolicy»	GRM::Scheduling
	Processing Unit	«ComputingResource»	GRM::ResourceTypes
	Global Clock - scan	«LogicalClock» - clockTick	TimeAccesses::Clocks

4 CPU Usage Analysis through Constraint Optimization

Figure 4 shows an overview of our solution for CPU usage analysis using constraint optimization. Our solution has four main elements: (1) time and concurrency information, (2) scheduling variables, (3) objective functions, and (4) constraints characterizing schedulability algorithms.

Intuitively, given the input (time and concurrency information), the goal is to compute values for the scheduling variables such that: (a) the schedulability constraints are satisfied, and (b) an objective function is maximized or minimized depending on the problem at hand. One main advantage of approaching our objectives as a constraint optimization problem is that such computations can be performed using off-the-shelf constraint optimization tools. We ground our formulation of the CPU usage problem on the COMET tool. This choice is motivated mainly by the efficient implementation of complete search in COMET and its support for parallel search (see Section 5), which is used for the evaluation of our approach in this paper. Below, we discuss each of the four main elements of our solution and outline their implementation in COMET.

(1) Time and Concurrency Information. All the input data in Fig 4 (part (1)) corresponds to the elements in our conceptual model in Section 3, and hence can be automatically extracted from the UML/MARTE models. We implement this information using COMET pre-defined data types. We define the notion of *observation time interval* as the time we spend observing the thread executions and denote it by T.

(2) Scheduling Variables. These variables specify a *schedule* for a given set of activities A during an observation time interval T. Specifically, a schedule specifies the actual start time $start(a)$ and the actual end time $end(a)$ for every activity $a \in A$. We denote the duration of an activity a by $d(a)$ (not to be confused with $delay$ which represents the delay time after activities). In non-preemptive scheduling, $d(a)$ is simply defined as

```
// (1) Time and concurrency information (Input)

range Threads = 0..n-1;  range Activities = 0..m-1;
int c = 3;                          // Number of cores
int p[Threads] = ..;                // Periods
int priority[Threads] = ..;         // Priority
  ....

// (2) Scheduling variables (Output)
var{int} start[Activities];                  // Actual start times
var{int} end[Activities];                    // Actual end times
var{int} active[Activities, T];              // Active matrix for individual time points
var{int} eligible_for_execution[Activities]; // Start times in ideal situation

// (3) Objective function (maximizing CPU usage)
maximize
  sum (a in Activities, t in T) (active[a, t])/ c* sizeofT    // CPU usage computation function

// (4) Constraints (characterizing schedulability algorithms)
subject to
{
  forall (a in Activities )
       post (end[a] < p[thread(a)]);  // An activity must end before the period of its thread
       post (active[a, start[a]] == 1);  // ...
       ....
}
```

Fig. 4. CPU usage as a constraint optimization problem. The full COMET implementation can be found at [12].

the length of the interval between $start(a)$ and $end(a)$, i.e., $d(a) = end(a) - start(a)$. But for preemptive scheduling, a can be interrupted during its execution, and hence, it may not be executing continuously. Therefore, we define $d(a)$ to be a set variable representing the set of time points at which a executes. In addition, for an activity a and time point t, we define a function $active(a, t)$ as follows:

$$active(a, t) = \begin{cases} 1 \text{ if } a \text{ executes at time } t \\ 0 \text{ otherwise.} \end{cases}$$

To account for multi-core scheduling, we define a variable $eligible\_for\_execution(a)$, or $efe(a)$ for short, that returns the earliest possible time that a can start running assuming that the number of cores is infinite, and hence, there is no bound on the number of activities that can run in parallel.

We implement scheduling variables as COMET variables with a specific type and a finite range. Values for these variables are computed within a given observation time interval T. In our formulation, we have added a new dimension to the scheduling variables to compute these variables for multiple execution rounds, where the number of rounds is determined by T (not shown in Figure 4 to avoid clutter, see [12]).

(3) Constraints. We use first-order logic to express the constraints. All the constraints are provided below. We omitted constraint formulations when the formulations were straightforward or lengthy. The complete formulations are available at [12].

▷ **Well-formedness (sanity rules).**

- Every activity must finish before the period of its corresponding thread elapses and cannot start before the start time of that thread.
- The number of time points at which an activity is running is bounded by its min/max duration.
- An activity starts running at its start time, ends just before its end time, and does not run before its start time or after its end time.

▷ **Loop Threads.** Consider activities $a_0^i, \ldots, a_q^i$ representing the activities of iteration i of a thread. Then, for every iteration i, we must have: $start(a_0^{(i+1)}) \geq end(a_q^i)$.

▷ **Temporal Precedence.** For every $a, a' \in A$ s.t. $a \preceq_t a'$, we have $start(a') - end(a) \geq delay(a)$. Note that $delay(a) = 0$ if a is not followed by a delay.

▷ **Synch/Asynch Communication.** For every $a, a' \in A$ s.t. $a \preceq_d a'$, if the communication is synchronous then we have $start(a') \geq end(a)$.

▷ **Buffer.** For every $a, a' \in A$ s.t. $a \preceq_d a'$, if the communication goes through a shared buffer then if $start(a) < start(a')$, then $start(a') \geq end(a)$. This is because a locks the shared resource during its execution. Also, if a and a' access the same buffer (but no data dependency relation is known between them), then a and a' cannot be active at the same time at any given time.

▷ **Multi-Core.** The number of running activities at every time point is less than or equal to the number of cores:

▷ **Scheduling Policy.**

- Each activity can potentially be preempted: $\forall a \in A \cdot end(a) - start(a) \geq d(a)$.
- The earliest time an activity a can start ($efe(a)$) is after the arrival time of its corresponding thread and after the earliest termination time of all the activities *preceding* a. Here, precedence includes both temporal precedence ($\preceq_t$) and data dependency ($\preceq_d$) orderings.
- At any time, if there are two activities that can be scheduled for execution in parallel but only one is running, the one that is not running has a lower priority.

Amongst the constraints above, only the scheduling policy constraints have a context-specific nature and need to change according to the specific policy used in a given system. The remaining constraints are generic and be reused across different domains and applications.

(4) Objective Functions. Our objective is to find combinations of input values that can generate schedules that consume the CPU time most, and hence, are more likely to violate CPU usage requirements of the system. To capture high usage of CPU time, we define two alternative objective functions. The first one computes average CPU usage, is denoted by f_{usage}, and is defined as:

$$f_{usage} = \frac{\sum_{a \in A, t \in T} active(a,t)}{T \times c}$$

The summation $\sum_{a \in A, 0 \leq t \leq T} active(a, t)$ measures the total time points when at least one activity is running, and $T \times c$ is the total available time on all the cores. The second objective function, called *makespan*, measures the total length of the schedule. We denote this objective function by $f_{makespan}$ and define it as:

$$f_{makespan} = \max_{a \in A} end(a)$$

The $f_{makespan}$ function is the time it takes for all the activities in an application to terminate after the arrival time of the first thread in that application. Makespan is a common metric for measuring response time [7].

By maximizing either f_{usage} or $f_{makespan}$, we compute schedules that are more likely to violate the CPU usage requirements. Note that f_{usage} or $f_{makespan}$ are heuristics as their accuracy is bounded by the accuracy of the input data and the precision of our constraints in characterizing the domain. Therefore, these functions should not be viewed as measures for the actual CPU usage of the system. In Section 5, we discuss how the input values maximizing these functions can be used to generate test cases for CPU usage requirements.

5 Evaluation

The main goal of our evaluation is to investigate whether our technique can effectively help engineers in deriving test cases for CPU usage requirements. The practical usefulness of our approach depends on (1) whether the input to our approach can be provided with reasonable overhead, and (2) whether the engineers can utilize the output of our analysis to derive test cases that can maximize CPU usage.

(i) Prerequisite and Overhead. As discussed before, the information required for CPU usage estimation is captured by the conceptual model in Figure 2. To gather this information, we first built UML sequence diagrams for the IO drivers in our partner company using the existing design and implementation of the drivers. The resulting sequence diagrams were iteratively validated and refined in collaboration with the lead engineer of the IO drivers. Sequence diagrams are popular for visualizing concurrent multi-threaded interactions and are intuitive to most developers as was confirmed in our industry collaboration [13].

The quantitative elements in Figure 2 for our case study were obtained as follows: The values for priority and period of the threads, and the size of the buffers were extracted from the certification design documents and the IO driver code. The min/max inter-arrival times of **IODispatch**, which is an aperiodic thread, and the values for the min/max duration of activities in our case study were extracted from the performance profiling logs of the drivers. We created the sequence diagrams augmented with the timing information over 8 days, involving approximately 25 man-hours of effort. This was considered worthwhile as such drivers have a long lifetime and are regularly certified.

Finally, we obtained the computing platform information in Figure 2 from the RTOS configuration and hardware design documents. Note that, given the mapping in Table 1, any modeling development environment that supports UML/MARTE can be used to develop and manipulate our input design notation.

(ii) Test Case Derivation. The I/O drivers in our study are subject to certification based on the IEC61508 standard [14], which is one of the most detailed and widely-used functional safety standards. It specifies 4 levels of safety, called *Safety Integrity Levels (SILs)*. SIL1 is the lowest and SIL4 is the highest level. The drivers in our study need to be compliant to IEC61508 up to SIL2 or SIL3 depending on the context of their application. Stress testing (subjecting the system to harsh inputs with the intention of breaking it [6]) is classified by IEC61508 as "Recommended" for SILs 1-2 and "Highly Recommended" for SILs 3-4. "Highly Recommended" techniques/measures are often seen as "mandatory" by the certifiers, unless the supplier provides a convincing argument as to why a highly recommended technique/measure does not apply. Subsequently, the engineers in our partner company needed to stress test the drivers (mandatory for SIL3 deployments).

We characterize the stress test cases in our case study by the delay times that (potentially) follow execution of activities, i.e., the delay attribute of the activity class in Figure 2. IO drivers are instantiated in different environments with different numbers and kinds of detectors and FMSs. The delay times after the IO driver activities must be set to values that match the load and speed of the detectors and FMSs.

Based on the engineers' intuition, large and complex hardware configurations, e.g., those consisting of several thousands of detectors, are more likely to violate the CPU

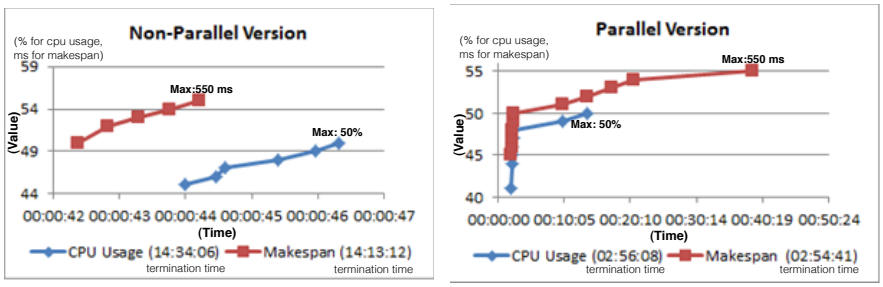


Fig. 5. The result of maximizing $f_{makespan}$ and f_{usage} (Section 4) for both parallel and non-parallel COMET implementations.

usage requirements. To identify the suspicious hardware configurations, however, the analysis provided in this paper is necessary because the hardware configurations affect the delay times of the IO drivers activities, and subsequently, the CPU usage estimates.

For example, the size of the delay time at step 3 of the data transfer scenario in Figure 1(b) can heavily impact the CPU usage. Specifically, the delay time cannot be so small that **IODispatch** (Figure 1(a)) keeps the CPU busy for so long that it exceeds the given CPU usage requirement. Neither can the delay be too large, because then **pullData**, which is periodic, may miss its deadline. Specifically it may quickly fill up the **Message Box 1** buffer, which in turn causes **pullData** to be blocked and waiting for **IODispatch** to empty **Message Box 1**, which is now very slow due to a large delay time. As a result, **pullData** may not be able to terminate before its next scan arrival.

To derive stress test cases based on the delay times of the activities, in our formulation in Figure 4, we specify $delay$ as an output variable whose value is bounded within a range. The search then varies the values of these variables to maximize $f_{makespan}$ and f_{usage}. Those combinations that maximize our objective functions are more likely to stress the system to the extent that the CPU usage requirements are violated.

To perform the above experiment, we implemented the constraint optimization formulation in Figure 4 in COMET Version 2.1.0 [7]. We further used the native support of COMET for parallel programming to create a distributed version of our COMET implementation that divides the search work-load among different cores. To perform the experiment, we varied the observation time T from 1s to a few seconds and set the quantum time (i.e., the minimum time step that a scheduler may preempt activities) to 10 ms. The input model included eight activities belonging to three parallel threads.

Figure 5 shows the result of our experiment, maximizing $f_{makespan}$ and f_{usage} for both parallel and non-parallel COMET implementations. In both diagrams, the X-axis shows the time, and the Y-axis shows the size of $f_{makespan}$ in ms, and the percentage for f_{usage}. In our experiment, we used a complete (exhaustive) constraint solver of COMET, and ran it on a MacBook Pro with a 2.0 Ghz quad-core Intel Core i7 with 8GB RAM. As shown in the figure, the search terminated in both cases: after around 14 hours for the non-parallel version, and after around 2 hours and 55 min for the parallel version. The maximum computed values are: 50% for f_{usage}, and 550 ms for $f_{makespan}$. In the non-parallel case, the maximum result was computed after around 1 hour and 10 min for $f_{makespan}$, and 1 hour and 13 min for f_{usage}. No higher value was found in the remainder of the search which took more than 14 hours in total. In the parallel case,

it took about 15 min to find the maximum for f_{usage}, and 40 min to compute that for $f_{makespan}$. To make sure these values were indeed maximum, the search continued until it terminated after 2 hours and 55 min.

In the end, we could compute maximum values for $f_{makespan}$ and f_{usage} in around 2 hours and 55 min using COMET's built-in support for parallel search. The values for the delay times maximizing $f_{makespan}$ and f_{usage} are candidates for stress test cases. We have recorded these values and have communicated them to our partner company.

Currently, the engineers at our partner company spend several days simulating their systems and monitoring the CPU usage without following a systematic strategy for stressing the systems to their CPU usage limits. We expect that by executing their systems based on the values produced by our approach, they can push the systems to states where the CPU usage is maximized and ensure that the input delay times remain within safe margins. The engineers at our partner company intend to test their system using our findings. Our experimental results and the input data values are available at [12].

6 Related Work

All approaches to performance engineering and schedulability analysis require a model of the time and concurrency aspects of the system under analysis [15]. Examples of such modeling languages include queuing networks [16], stochastic Petri nets [17], and stochastic automata networks [18]. Recently, there has been a growing interest in developing standardized languages to enhance the adoption of performance engineering concepts and techniques in the industry [19]. The most notable these languages is MARTE which extends UML with concepts for modeling and quantitative analysis of real-time embedded systems [5]. While a UML-profile, MARTE also encompasses the timing and concurrency abstractions in many other languages, e.g., Architecture Analysis and Design Language (AADL) [20]. As indicated by the mapping from our conceptual model to MARTE (Section 3.2), the abstractions we use in our work already exist in MARTE. However, MARTE is a large profile and by itself does not provide guidelines on what subset of it is required for a particular type of analysis. Our conceptual model can thus be viewed as a subset selection of MARTE, aimed specifically at CPU usage analysis.

The techniques for analysis of real-time systems can be divided into two general groups: (1) *Approaches based on real-time scheduling theory [9].* These approaches estimate schedulability of a set of tasks through customized formulas and theorems that often assume worst case situations only such as worst case execution times, worst case response times, etc. Their results, therefore, can be too conservative because due to inaccuracies in estimating worst-case time values, the worst-case situations may never happen in practice. As a result, in general, we cannot rely on schedulability theory alone when dealing with analysis of real-time systems. Moreover, extending these theories to multi-core processors has shown to be a challenge [21, 22].

(2) *Model-based approaches to schedulability analysis.* The idea is to base the schedulability analysis on a system model that captures the details and specifics of real-time tasks. This provides the flexibility to incorporate specific domain assumptions and a range of possible scenarios, not just the worst cases [23, 24]. Most approaches that fall in this category, including our work, can deal with multi-core processors as well [24].

We formulate the problem of CPU usage analysis as a constraint optimization problem. Our work is inspired by *Job shop scheduling* – a well-known optimization problem

where jobs are assigned to resources at particular times [25]. Job shop has several variations. Our formulation is closest to the *discrete resources* variant [7], but differs from it in that we need to specify scheduling policies used by the underlying RTOS.

Model checkers, in particular, real-time model checkers, e.g., UPPAAL [26], have been successfully used for the evaluation of time-related properties. Model checking is intended to be used for *verification*, i.e., to check if a given set of real-time tasks satisfy some property of interest. To adapt model checkers to checking different properties of real-time applications, the underlying state machines are built such that the question at hand can be formulated as a reachability query. For example, in [24], in order to analyze CPU-time usage, an *idle* state-machine is added to the set of interacting timed-automata to keep track of the CPU-time, and the error states were chosen so that their reachability could lead to violation of the CPU-time usage limit.

In our work, the property to be checked is captured by a *quantitative* objective function as opposed to a *boolean* reachability property, as in the case of model checking. Therefore, our work is more geared towards *optimization* with applications in test-case generation rather than verification. One significant practical advantage is that, to adapt our formulation to check other kinds of real-time properties, it often suffices to change the objective function, and most of the constraints remain untouched. Lastly, to handle multiple cores, the existing UPPAAL-based solution in [22] assumes that a mapping between threads and cores is given a priori. Our approach in contrast does not require any mapping between threads and cores.

7 Conclusions and Future Work

We provided a practical approach to support derivation of stress test cases for the CPU usage requirements of concurrent embedded applications running on multi-core platforms. We proposed a conceptual model that captures, independently from any modeling notation, the abstractions required for analysis of CPU usage. We mapped our conceptual model onto the standard modeling language UML/MARTE to support the application of our approach in practice. We, then, formulated the CPU usage problem as a constraint optimization problem over our conceptual model, and implemented our formulation in COMET. Our evaluation on a real case study shows that our approach (1) can be applied with practically acceptable overhead, and (2) can identify test cases that maximize CPU usage. These (stress) test cases are crucial for building satisfactory evidence to demonstrate that no safety risks are posed by potential CPU overloads. Finally, we note that while our approach cannot provide proofs, we can always provide results given a (partial) set of declarative constraints and within a time budget.

Our solution draws on a number of context factors (Section 2) which need to be ascertained before our solution can be applied. While the generalizability of these factors need to be further studied, we have found the factors to be commonplace in many industry sectors relying on embedded systems. In the future, we plan to perform larger case studies to better evaluate the generalizability and scalability of our approach and experiment with other search methods, in particular, meta-heuristic search methods and hybrid approaches combining complete and meta-heuristic search strategies.

Acknowledgments. *We thank Bran Selic for his useful comments on an earlier draft of this paper. We thank the research council of Norway for partially funding this work. L. Briand was supported by a FNR PEARL grant.*

References

1. Jackson, D., Thomas, M., Millett, L., eds.: Software for Dependable Systems: Sufficient Evidence? National Academy Press (2007)
2. Henzinger, T., Sifakis, J.: The embedded systems design challenge. In: FM. (2006) 1–15
3. Lee, E., Seshia, S.: Introduction to Embedded Systems: A Cyber-Physical Systems Approach. http://leeseshia.org (2010)
4. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
5. A UML profile for MARTE: Modeling and analysis of real-time embedded systems (May 2009)
6. Beizer, B.: Software testing techniques (2. ed.). Van Nostrand Reinhold (1990)
7. Hentenryck, P.V., Michel, L.: Constraint-Based Local Search. The MIT Press (2005) www.dynadec.com.
8. Wind River VxWorks. http://www.windriver.com/products/vxworks/ (2009)
9. Liu, J.W.S.: Real-time systems. Prentice Hall (2000)
10. Sprunt, B.: Aperiodic task scheduling for real-time systems. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1990)
11. OMG: The Unified Modelling Language. Version 2.1.2. http://www.omg.org/spec/UML/2.1.2/ (2007)
12. Alesio, S.D.: The CPU usage constraints in COMET. http://home.simula.no/~stefanod/comet.pdf
13. Harel, D., Marelly, R.: Specifying and executing behavioral requirements: the play-in/play-out approach. Software and System Modeling **2**(2) (2003) 82–107
14. IEC 61158: industrial communication networks - fieldbus specifications. International Electrotechnical Commission (2010)
15. Cortellessa, V., Marco, A.D., Inverardi, P.: Model-Based Software Performance Analysis. Springer (2011)
16. Lazowska, E., Zahorjan, J., Graham, S., Sevcik, K.: Quantitative system performance computer system analysis using queueing network models. In: Int. CMG Conference. (1984) 786–788
17. Marsan, M., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with generalized stochastic petri nets. SIGMETRICS Performance Evaluation Review **26**(2) (1998) 2
18. Plateau, B., Atif, K.: Stochastic automata network for modeling parallel systems. IEEE Trans. Software Eng. **17**(10) (1991) 1093–1108
19. Petriu, D. In: Software Model-based Performance Analysis. John Wiley & Sons (2010)
20. Hudak, J., Feiler, P.: Developing AADL models for control systems: A practitioner's guide (October 2006)
21. Bertogna, M.: Real-Time Scheduling Analysis for Multiprocessor Platforms. PhD thesis, Scuola Superiore Sant'Annna, Pisa (2007)
22. David, A., Illum, J., Larsen, K., Skou, A. In: Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1. CRC Press (2010) 93–119
23. Briand, L., Labiche, Y., Shousha, M.: Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. Genetic Programming and Evolvable Machines **7**(2) (2006) 145–170
24. Mikucionis, M., Larsen, K., Nielsen, B., Illum, J., Skou, A., Palm, S., Pedersen, J., Hougaard, P.: Schedulability analysis using UPPAAL: Herschel-Planck case study. In: Proceedings of 4th International Symposiumo on Leveraging Applications of Formal Methods, Verification and Validation; track: Quantitative Verification in Practice. (2010)

25. Applegate, D., Cook, W.: A computational study of the job-shop scheduling problem. INFORMS Journal on Computing **3**(2) (1991) 149–156
26. Behrmann, G., David, A., Larsen, K.: A tutorial on uppaal. In Bernardo, M., Corradini, F., eds.: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures. Volume 3185 of Lecture Notes in Computer Science., Springer Verlag (2004) 200–237