# Worst-case Scheduling of Software Tasks
## A Constraint Optimization Model to Support Performance Testing

Stefano Di Alesio[1,2], Shiva Nejati[2], Lionel Briand[2], and Arnaud Gotlieb[1]

[1] Certus Centre for Software Verification & Validation, Simula Research Laboratory, Norway
{stefano, arnaud}@simula.no
[2] Interdisciplinary Centre for Reliability, Security and Trust (SnT), University of Luxembourg, Luxembourg {shiva.nejati, lionel.briand}@uni.lu

**Abstract.** Real-Time Embedded Systems (RTES) in safety-critical domains, such as maritime and energy, must satisfy strict performance requirements to be deemed safe. Therefore, such systems have to be thoroughly tested to ensure their correct behavior even under the worst operating conditions. In this paper, we address the need of deriving worst case scenarios with respect to three common performance requirements, namely task deadlines, response time, and CPU usage. Specifically, we investigate whether this worst-case analysis can be effectively re-expressed as a Constrained Optimization Problem (COP) over the space of possible inputs to the system. Solving this problem means finding the sets of inputs that maximize the chance to violate performance requirements at runtime. Such inputs can in turn be used to test if the target RTES meets the expected performance even in the worst case. We develop an OPL model for IBM ILOG CP OPTIMIZER that implements a task priority-based preemptive scheduling, and apply it to a case study from the maritime and energy domain. Our validation shows that (1) the input to our model can be provided with reasonable effort in an industrial setting, and (2) the COP effectively identifies test cases that maximize deadline misses, response time, and CPU usage.

## 1 Introduction: Performance Testing in Safety-Critical Systems

Systems in domains such as avionics, automotive, and maritime are often safety-critical, implying that their failure could result in catastrophic consequences. For this reason, their safety-related software components are usually subject to third-party certification to be deemed operationally safe. In particular, software certification has to take into account performance requirements specifying how the system should execute on its hardware platform, and how it should react to its environment [15]. Such requirements often specify constraints on response time, jitter, task deadlines, and computational resources utilization [13, 20]. Widely used safety standards, such as IEC 61508 and IEC 26262, state that performance testing is highly recommended to provide an evidence that the system is safe [6]. However, safety-critical systems are progressively relying on real-time embedded software that features multi-threaded application design, highly configurable operating systems, and multi-core architectures for computing platforms [17]. The concurrent nature of embedded software also entails that the order of external events triggering the systems tasks is often unpredictable [12]. Such increasing software complexity renders performance testing more and more challenging. This

aspect is reflected by the fact that most existing software testing approaches target only the system functionality, even though the degradation in performance can have more severe consequences than mere incorrect behavior [27].

In this paper, we consider three common classes of performance requirements, concerning respectively hard real-time, soft real-time, and resource constraints. Specifically, (1) we relate the hard real-time constraints to *task deadlines* requirements, stating that the system tasks should always terminate before a given completion time. Such strict requirements entail that even a single deadline miss severely compromises the system operational safety. (2) For the soft real-time constraints, we consider *response time* requirements, stating that the system should respond to external inputs within a specified time. Failure to do so poses negative consequences over the Quality of Service (QoS). (3) Finally, we consider *CPU usage* requirements, stating that the system should always keep a certain percentage of free CPU. Limiting the CPU usage is a necessary safety precaution. Indeed, if the CPU usage trespasses a certain threshold, the system may fail to timely respond to safety-critical alarms.

To check whether the system satisfies these requirements, we need to identify worst-case scenarios with respect to deadline misses, response time, and CPU usage. Such scenarios are determined by the way tasks are scheduled to execute at runtime by the Real-Time Operating System (RTOS). The schedules in turn depend on real-time, unpredictable events, on constraints deriving from software design, and on the system execution platform. For instance, critical tasks in RTES are usually ready to be executed upon triggers that depend on the external environment. Furthermore, the system design constrains the way tasks interact with each other, specifying temporal relationships, and communication through shared resources with exclusive access. Finally, RTOS are in general configured with a priority-based scheduling policy, which entails that the lowest priority task must be preempted when a higher priority task is ready for execution. However, this preemption can only occur when the running task is not locking a shared resource, and is only necessary when there is no available processor core. This mix of real-time, software design, and execution platform constraints on task scheduling renders the analysis of runtime scenarios challenging in the context of RTES.

Our work focuses on performance testing, whose goal is to identify scenarios that exercise a system in a way to either violate performance requirements, or be as close as possible to doing so. Consistent with widely used terminology [5], we refer to this activity as *stress testing*. We propose a strategy to find combinations of system inputs that maximize the likelihood of violating performance requirements. Such input combinations are characterized by sequences of *arrival times* for aperiodic tasks in the target software system, and we refer to each sequence as a *stress test case*. Finding these test cases is not trivial, since the set of all possible arrival times for aperiodic tasks quickly grows as the system size increases. Therefore, it is practically impossible to investigate all the potential ways in which the arrival times can determine task schedules at runtime. This reason motivates the need of a systematic search that effectively finds stress test cases likely to reveal deadline misses, long response time, and high CPU usage.

**Contributions of this Paper.** The main contribution of this paper is to address the systematic generation of stress test cases by applying Constraint Programming. Specifically, we present a Constraint Optimization Model (COP) to automate the generation

of stress test cases, which is inspired by the work done in this field to solve traditional scheduling problems [4]. We evaluate our model using a practical application on a RTES from the maritime and energy domain. We cast the problem of generating stress test cases as an OPL model designed for the IBM ILOG CP OPTIMIZER, that models the system design, executing platform, and performance requirements. The OPL model in this paper builds on our previous work [10, 11, 22]. Specifically, we started [22] with a COMET optimization model, where we addressed the validation of CPU Usage and response time. Then [10], we devised an OPL version of the model where we focused on task deadlines. Finally [11], we included a dedicated search procedure for a smarter labeling of variables, and compared our constraint-based approach with metaheuristic search techniques. Our earlier work included a variable boolean matrix showing tasks execution over time, that proved to severely limit the efficiency of our model. In this work, we significantly improve the data structures representing task executions, and demonstrate the applicability of our new COP to an industrial case study. Specifically,

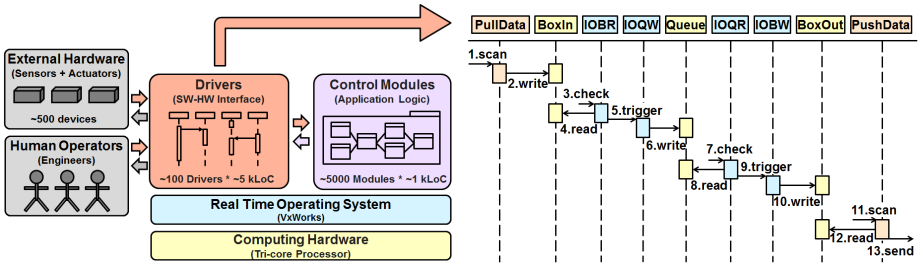
1. We provide a detailed OPL model which implements a task priority-based scheduling process by considering a discretized matrix, as opposed to a boolean one, which represents task executions over time. In addition, we address the search for solutions adapting the dedicated heuristic proposed in previous work [11].
2. We demonstrate the efficiency of our OPL model by applying it to an industrial case study representing a multi-threaded I/O driver with several instances running concurrently on a multi-core platform. Our approach successfully found scenarios violating the three performance requirements in a few minutes.

## 2 The Fire and gas Monitoring System

The main motivation behind our work comes from a case study of a Fire and gas Monitoring System (FMS) in the maritime and energy domain. The goal of the system is to monitor potential gas leaks in oversea oil extraction platforms, and trigger an alarm in case a fire is detected. The system displays to human operators data coming from smoke and heat detectors, and gas flow sensors. When the system receives critical data from the hardware sensors, it automatically triggers actuators, such as fire sprinklers and audio/visual alarms. An older version of this case study was presented in our previous work [22], where we discussed the detailed design and modeling of the I/O drivers. The FMS software architecture is shown in Figure 1a.

Drivers implement I/O communication between the control modules of the system, and the external environment, such as hardware sensors, actuators, and human operators. In the FMS, thousands of instances of I/O drivers run concurrently interacting with several hundreds sensors. The software components of the FMS are executed on a Real-Time Operating System that runs on a tri-core computing platform.

Drivers in the FMS share the same design pattern, featuring six tasks that communicate through three buffers which have fixed capacity and cannot be simultaneously accessed by different tasks. Figure 1b shows the typical operational scenario, that is a unidirectional data transfer between hardware sensors and control modules. (1) *PullData* periodically receives data from sensors or human operators, formats the data in an appropriate command form, and (2) writes it in the buffer *BoxIn*. (3) When *BoxIn* is full, the *check* signal activates *IOBoxRead* that (4) reads the data from the buffer and




(a) Software architecture of the FMS (b) Typical drivers data transfer scenario

Fig. 1: Description of the Fire and gas Monitoring System

(5) triggers *IOQueueWrite*. *IOQueueWrite* extracts the commands from the data, and (6) stores them in the priority *Queue*. When *Queue* reaches a critical capacity, (7) the *check* signal activates *IOQueueRead* that (8) reads the highest priority command and (9) triggers *IOBoxWrite* which in turn (10) writes the command to *BoxOut*. When the periodic *scan* signal (11) activates *PushData*, the task (12) reads the commands from *BoxOut* and finally (13) sends them to the control modules for processing.

The data transfer functionality is subject to strict performance requirements. Specifically, in each FMS driver, (1) no task should miss its deadline, (2) the response time should be less than one second, and (3) the average CPU usage should be below 20%. The main variables determining whether or not these requirements will be satisfied at runtime are the arrival times of the *check* signal. These arrival times depend on the external environment, in the sense that depend on the data sent by the hardware sensors via *PullData*. The arrival times also vary across different system executions, as a consequence of the impossibility to predict the data coming from the sensors. Therefore, in order to evaluate task deadlines, response time, and CPU usage, we need a strategy to search all the possible task arrival times. This search has the objective of finding scenarios that are predicted to violate the requirements, or be close to violating them. Indeed, the more likely a scenario is predicted to violate a performance requirement, the higher the chances that the test case characterized by such scenario will stress the system.

## 3 Related Work

Testing multi-threaded concurrent software has largely focused on functional properties, rather then system performance [27]. Specific methods [12] for design-time performance analysis have been proposed to estimate the schedulability of a set of tasks through formulas and theorems from the Real-Time Schedulability Theory [26], or with model checking techniques [2]. In our experience, performance analysis is addressed in industry mainly with Performance Engineering, which extensively relies on profiling and benchmarking tools to dynamically analyze performance properties [16]. Such tools, however, are limited to producing a small number of system executions, and require their manual inspection. Performance analysis can check the overall sanity of the system performance, but cannot replace systematic stress testing.

For performance testing, search-based approaches have extensively been used [1], especially in the domain of distributed systems. Specifically, Genetic Algorithms (GA) have successfully been used to support performance testing, in particular with respect

to QoS constraints [24] or computational resources consumption [7]. GA have also been used to generate test cases for testing tasks timeliness [23]. As for hard real-time properties such as deadline misses, the state-of-the-art is represented by the work of Briand et al. [8], that we used as a baseline for comparison in our previous work [11].

For schedulability analysis, CP approaches [4] have been studied for long time, especially in the domain of job-shop scheduling problems [19]. Among those, several approaches target task real-time constraints such as task deadlines [14], or timeliness [21]. Preemptive scheduling problems have also been solved both with pure CP [9], and with hybrid approaches featuring combinations with GA [28]. Furthermore, recent implementations [18] have successfully used IBM ILOG CP OPTIMIZER and OPL for scheduling problems, albeit not addressing task preemption.

Despite the extensive literature for constraint based scheduling, we are unaware of CP approaches targeted to test case generation, such as the generation of worst case scenarios, and of approaches addressing all the complexities of RETS such as multi-core architectures, task dependencies, aperiodic tasks, and preemptive scheduling policies.

## 4 Supporting Performance Testing: A New Application of COPs

We address the problem of determining worst-case schedules of tasks with an approach inspired by the work done in Constraint Programming to solve traditional scheduling problems [4]. Specifically, we cast the search for real-time properties that characterize the worst-case schedules, namely arrival times for aperiodic tasks, as a Constraint Optimization Problem (COP). The key idea behind our formulation relies on five main points. (1) First, we model the system design, which is static and known prior to the analysis, as a set of constants. The system design mainly consists of the tasks of the real-time application, their dependencies, period, duration, deadline, and priority. Constants of our model are described in Section 4.3. (2) Then, we model the system properties that depend on runtime behavior as a set of variables. The main real-time properties are the number of task executions, the arrival times of aperiodic tasks, and the specific runtime schedule of the tasks. (3) We model the RTOS scheduler as a set of constraints among such constants and variables. Indeed, the real-time scheduler periodically checks for triggering signals of tasks and determines whether tasks are ready to be executed or need to be preempted. (4) We model the performance requirement to be tested (i.e., task deadlines, response time, or CPU usage) as an objective function to be maximized. (5) Finally, we encapsulate the logic behind the RTOS scheduler in an effective labeling strategy over the variables of the model. By design, the scheduler tries to execute high priority tasks as soon as possible, potentially preempting tasks with lower priority. We exploit this behavior by proposing a labeling strategy for the variables related to tasks execution. Our analysis is subject to two main assumptions:

1. The RTOS scheduler checks the running tasks for potential preemptions at regular and fixed intervals of time, called *time quanta*. Therefore, each time value in our problem is expressed as a multiple of a time quantum. Accordingly to the specification of the RTOS executing the FMS, we will consider the length of ten milliseconds for time quanta.
2. The interval of time in which the scheduler switches context between tasks is negligible compared to a time quantum.

These two assumptions are reasonable in the context of RTES, as the scheduling rate of operating systems varies in the ranges of few milliseconds, while the time needed for context switching is usually in the order of nanoseconds [25]. These assumptions allow us to consider time as discrete, and model the COP as an Integer Program (IP) over finite domains. We implemented the COP in OPL, and solved it with IBM ILOG CPLEX CP OPTIMIZER. This choice was motivated by practical reasons, such as extensive documentation, strong supporting community, and its acknowledged efficiency to solve optimization problems. Despite the scheduling nature of our problem, we implemented our model as a traditional IP as opposed to using the scheduling features of OPL and CP OPTIMIZER. This is because we could not express a preemptive priority-driven scheduling behavior in an effective way that exploited the capabilities of the solver.

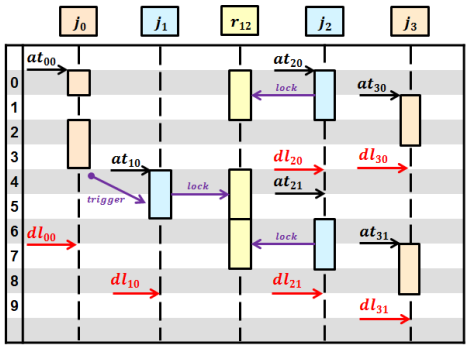



Fig. 2: Real-time scheduling example of four tasks on a dual-core platform

The rest of this section details our constraint model using the example shown in Figure 2. This system features four tasks in increasing priority order, $j_0$ to $j_3$, running on a dual-core platform for 10 time units. $j_0$ and $j_1$ are executed once, while $j_2$ and $j_3$ are executed twice. The figure reports the arrival times and deadlines of the tasks, respectively labeled by $at$ and $dl$, where the first index represents the task, and the second the task execution. In this example, $j_0$ is aperiodic, while $j_2$ and $j_3$ are periodic. Note that task $j_1$ is triggered by $j_0$ upon termination, and that $j_1$ and $j_2$ share the resource $r_{12}$ with exclusive access.

### 4.1 Constants

Constants are implemented as integers (*int*), integer ranges (*range*), tuples (*tuple*), sets of tuples (*setOf*) and integer expressions. Integers values are defined as external data.
**Observation Interval.** Let $T$ be an integer interval of length $tq$, i.e., $T \stackrel{\text{def}}{=} [0, tq - 1]$, representing the time interval during which we observe the system behavior. $T$ is an integer interval, as a consequence that time is discretized in our analysis. Therefore, each time value $t \in T$ is a time quantum. In Figure 2, $tq = 10$ and $T = [0, 9]$.
**Number of platform cores.** Let $c$ be the number of cores in the execution platform. Therefore, $c$ represents the maximum number of tasks that can be executed in parallel. In Figure 2 $c = 2$, as at most two tasks are allowed to run in parallel.
**Set J of tasks.** Let $J$ be the set of tasks of the system. Each task $j \in J$ has a set of static properties, defined as constants, and a set of dynamic properties, defined as variables.

Let $J_p$, $J_a$, and $J_g$ respectively be the set of periodic, aperiodic, and triggered tasks of the system. $J_p$, $J_a$, and $J_g$ define a partition over $J$. We assume that OS tasks have lower priority than system tasks and can be preempted at any time, and hence, can be abstracted away in our analysis. Each task $j$ is implemented as an OPL tuple named *Task*, whose fields are the following non-relation constants. $J$ is implemented as an OPL tuple set, while $J_p$, $J_a$, and $J_g$ are OPL generic sets derived from $J$. In Figure 2, $J = \{j_0, j_1, j_2, j_3\}$, $J_a = \{j_0\}$, $J_p = \{j_2, j_3\}$, and $J_g = \{j_1\}$.

**Priority of tasks.** Let $pr(j)$ be the priority of task $j$. For simplicity, we define the set $HP_j$ of tasks having higher or equal priority than $j$: $HP_j \stackrel{\text{def}}{=} \{j_1 \in J \mid j \neq j_1 \wedge pr(j_1) \geq pr(j)\}$. In Figure 2, $pr(j_0) = 0$, $pr(j_1) = 1$, $pr(j_2) = 2$, and $pr(j_3) = 3$.

**Period of tasks.** Let $pe(j)$ be the period of the task $j$, only defined if $j$ is periodic. In Figure 2, $pe(j_2) = 5$ and $pe(j_3) = 6$.

**Offset of tasks.** Let $of(j)$ be the offset of the task $j$, i.e., the time, counted from the beginning of $T$, after which the first period of task $j$ begins. $of$ is only defined if $j$ is periodic. In Figure 2, $of(j_2) = 0$ and $of(j_3) = 1$.

**Minimum and maximum inter-arrival times of tasks.** Let $mn(j)$ and $mx(j)$ respectively be the minimum and maximum inter-arrival times of task $j$, i.e., the minimum and maximum time separating two consecutive arrival times of $j$. $mn(j)$ and $mx(j)$ are only defined if $j$ is aperiodic since for all periodic tasks $j$, $mn(j) = mx(j) = pe(j)$ trivially holds. In Figure 2, we assumed $mn(j_0) = 5$ and $mx(j_0) = 10$.

**Duration of tasks.** Let $dr(j)$ be the estimated Worst Case Execution Time (WCET) of task $j$. For simplicity, we define the integer interval $P_j$ of *execution slots* as $P_j \stackrel{\text{def}}{=} \big[0,\ dr(j) - 1\big]$. Since OPL does not support indexed ranges, $P_j$ is implemented as a single range $P \stackrel{\text{def}}{=} [0, \max_{j \in J}\ dr(j) - 1]$. This definition entails that $\forall j \in J \ \cdot \ P_j \subseteq P$. The iteration through values in $P_j$ is emulated with a logic implication. Indeed, the following properties hold for every logic predicate $C$ and arithmetic expression $E$:

$$\forall p \in P_j \ \cdot \ C(p) \iff \forall p \in P \ \cdot \ p < dr(j) \implies C(p) \tag{1}$$

$$\sum_{p \in P_j} E(p) = \sum_{p \in P} \big(p < dr(j)\big) \cdot E(p) \tag{2}$$

Note that in Equation 2 $\big(p < dr(j)\big)$ is a boolean expression that is true if $p < dr(j)$, and false otherwise. For the rest of this paper, equalities and inequalities written within parentheses represent boolean expressions that evaluate to the integer 1 if true, and to the integer 0 if false. This is also the default behavior in CP OPTIMIZER. In Figure 2, $dr(j_0) = 3$ and $P_{j_0} = [0, 1, 2]$.

**Deadline of tasks.** Let $dl(j)$ be the time, with respect to its arrival time, before which task $j$ should terminate. In Figure 2, $dl(j_0) = 7$, $dl(j_1) = 6$, $dl(j_2) = 4$, and $dl(j_3)$=3.

**Triggering relation between tasks.** Let $tg(j_1, j_2)$ be a binary relation between tasks $j_1$ and $j_2$ that holds if the event triggering $j_2$ occurs when $j_1$ finishes its execution. The relation $triggers$ is defined as irreflexive and antisymmetric. For simplicity, we respectively define the sets $TS_j$ and $ST_j$ of tasks triggered by and triggering $j$: $TS_j \stackrel{\text{def}}{=} \{j_1 \in J \mid tg(j, j_1)\}$ and $ST_j \stackrel{\text{def}}{=} \{j_1 \in J \mid tg(j_1, j)\}$. $tg$ is implemented as an OPL tuple with two fields, the first being the task triggering, and the second being the task triggered. In Figure 2, $tg(j_0, j_1)$ holds.

**Dependency relation between tasks.** Let $de(j_1, j_2)$ be a binary relation between tasks $j_1$ and $j_2$ that holds if there exists a computational resource $r$ such that $j_1$ and $j_2$ access $r$ during their execution in an exclusive way. This implies that $j_1$ and $j_2$ cannot be executed in parallel nor can preempt each other, but one can execute only after the other has released the lock on the resource. The relation $dependent$ is defined as reflexive and symmetric. For simplicity, we define the set $DS_j$ of tasks depending on $j$: $DS_j \stackrel{\text{def}}{=} \{j_1 \in J \mid j \neq j_1 \wedge de(j_1, j)\}$. $de$ is implemented as an OPL tuple with two fields, each being one of the task depending on the other. In Figure 2, $de(j_1, j_2)$ holds.

### 4.2 Variables

Independent variables in our model are implemented as OPL finite domain variables (*dvar int*). Dependent variables are implemented as OPL variable expressions (*dexpr int*) defined through equality constraints. The first three variables described hereafter, namely the number of task executions, their arrival times, and active sets, are independent variables. The remaining variables described in this section are all dependent.

**Number of task executions.** Let $te(j)$ be the number of times task $j$ is executed within $T$. For simplicity, we define the integer interval $K_j$ of task executions for the task $j$ as $K_j \stackrel{\text{def}}{=} [0,\ te(j) - 1]$. Furthermore, we refer to the $k^{\text{th}}$ execution of task $j$ as the couple $(j, k)$. We assume the minimum and maximum inter-arrival times bound the number of executions of an aperiodic task. This means that, for aperiodic tasks, $te(j)$ is defined as a variable with domain $\left[\left\lfloor \frac{tq}{mx(j)} \right\rfloor,\ \left\lfloor \frac{tq}{mn(j)} \right\rfloor\right]$. Similarly, we assume that offset and period statically determine the number of executions of periodic tasks so that $te(j) = \left\lfloor \frac{tq - of(j)}{pe(j)} \right\rfloor$. Therefore, the number of task executions of periodic tasks is constant, rather than variable. However, we do not formally distinguish it from the number of task execution for aperiodic tasks. $te$ is implemented as an integer array ranging over $J_p$ if the task is periodic (or ranging over $J_g$ if triggered by a periodic task), and as an integer variables array ranging over $J_a$ if the task is aperiodic (or ranging over $J_g$ if triggered by an aperiodic task). Since OPL does not support ranges with variable bounds, $K_j$ is implemented as a single constant range $K$:

$$K \stackrel{\text{def}}{=} \left[0,\ \max\left(\max_{j \in J_p} \left\lfloor \frac{tq - of(j)}{pe(j)} \right\rfloor, \max_{j \in J_a} \left\lfloor \frac{tq}{mn(j)} \right\rfloor\right)\right]$$

Note that $K$ is defined as a range from 0 to the largest upperbound for task executions of periodic and aperiodic tasks. This definition entails that $\forall j \in J \cdot K_j \subseteq K$. The iteration through values in $K_j$ is performed in a similar way as the case of $P_j$, thanks to the following properties for each logic predicate $C$ and arithmetic expression $E$:

$$\forall k \in K_j \cdot C(k) \iff \forall k \in K \cdot k < te(j) \implies C(k) \tag{3}$$

$$\sum_{k \in K_j} E(k) = \sum_{k \in K} \big(k < te(j)\big) \cdot E(k) \tag{4}$$

In Figure 2 $te(j_0) = 1$, $te(j_3) = 2$, $K_{j_1} = [0]$, and $K_{j_2} = [0, 1]$.

**Arrival time of task executions.** Let $at(j, k)$ be the time when an event notifies the RTOS that task $j$ should be executed for the $k^{\text{th}}$ time. We say that $j$ *arrives* for the $k^{\text{th}}$

time at time $t$ iff $at(j,k) = t$. When the specific execution $k$ of $j$ is understandable from the context, we will simply say that $j$ arrives at time $t$. In our analysis, we assume that the arrival time of periodic tasks is constant: $\forall j \in J_p,\ k \in K_j \ \cdot\ at(j,k) = of(j) + k \cdot pe(j)$. Similarly to the case of $te$, we do not formally distinguish the arrival times of periodic and aperiodic tasks. $at$ has domain $T$ for aperiodic tasks. In Figure 2, $at(j_0,0) = 0$ and $at(j_2,1) = 5$.

**Active set of task executions.** Let $ac(j,k,p)$ be the $p^{\text{th}}$ time quantum in $T$ in which task $j$ is running for the $k^{\text{th}}$ execution. We refer to the set of all $ac$ variables as the *schedule* produced by the arrival times of the tasks in $J$. $ac$ variables have domain $T$. In Figure 2, $ac(j_0,0,0) = 0$, $ac(j_0,0,1) = 2$.

**Preempted set of task executions.** Let $pm(j,k,p)$ be the number of time quanta for which the $k^{\text{th}}$ execution of task $j$ is preempted for the $p^{\text{th}}$ time: $pm(j,k,p) \stackrel{\text{def}}{=} ac(j,k,p) - ac(j,k,p-1) - 1$. $pm$ is only defined for $p > 0$. In Figure 2, $pm(j_0,0,1) = 1$, and $pm(j_0,0,2) = 0$.

**Start and end times of task executions.** Let $st(j,k)$ and $en(j,k)$, respectively be the first and the one after the last time quantum in which task $j$ is executing for the $k^{\text{th}}$ time. We say that $j$ *starts* or *ends* for the $k^{\text{th}}$ time at time $t$ iff respectively $st(j,k) = t$ or $en(j,k) = t - 1$. By definition, $st(j,k) \stackrel{\text{def}}{=} ac(j,k,0)$ and $en(j,k) \stackrel{\text{def}}{=} ac\big(j,k,dr(j) - 1\big) + 1$. In Figure 2, $st(j_0,0) = 0$ and $en(j_1,0) = 6$.

**Waiting time of task executions.** Let $wt(j,k)$ be the time that $j$ has to wait after its arrival time before starting its $k^{\text{th}}$ execution. By definition, $wt(j,k) \stackrel{\text{def}}{=} st(j,k) - at(j,k)$. In Figure 2, $wt(j_0,0) = 0$, and $wt(j_2,1) = 1$.

**Deadline of task execution.** Let $ed(j,k)$ be the absolute deadline of the $k^{\text{th}}$ execution of $j$, i.e., the time, with respect to the beginning of $T$, before which $j$ should terminate to meet its deadline. By definition, $ed(j,k) \stackrel{\text{def}}{=} at(j,k) + dl(j) - 1$. $ed$ is implemented as two-dimensional array of integer variable expressions ranging over the set $J$ and the range $K$. In Figure 2, $ed(j_0,0) = 6$, and $ed(j_1,0) = 8$.

**Deadline miss of task execution.** Let $dm(j,k)$ be the amount of time by which $j$ missed its deadline during its $k^{\text{th}}$ execution. By definition, $dm(j,k) \stackrel{\text{def}}{=} en(j,k) - ed(j,k) - 1$. $dm$ is implemented as two-dimensional array of integer variable expressions ranging over the set $J$ and the range $K$. In Figure 2, $dm(j_0,0) = -3$.

**Blocking task execution time quantum.** Let $bl(j,k,j_1,k_1,p_1)$ be a boolean variable that is true if in the interval $\big[at(j,k), st(j,k)\big)$ the task execution $(j_1,k_1)$ is active at the time slot $p_1$:

$$bl(j,k,j_1,k_1,p_1) \stackrel{\text{def}}{=} at(j,k) \leq ac(j_1,k_1,p_1) < st(j,k)$$

In Figure 2, $bl(j_2,1,j_1,0,1) = true$, since $(j_2,0)$ waits at $t = 5$ for the last time quantum of $(j_1,0)$ before starting.

**Higher priority active tasks.** Let $ha(j,k)$ be the number of time quanta in the interval $\big[at(j,k), st(j,k)\big)$ where exactly $c$ tasks having higher priority of $j$ and not depending on $j$ are active. Consider the summation indexes $j_1$, $k_1$, $p_1$ respectively defined in the sets $HP_j \setminus DS_j$, $K_{j_1}$, and $P_{j_1}$, and the summation indexes $j_2$, $k_2$, $p_2$ respectively defined in the sets $HP_j \setminus DS_j$, $K_{j_2}$, and $P_{j_2}$. We define:

$$ha(j,k) \stackrel{\text{def}}{=} \sum_{j_1,k_1,p_1} \left( bl(j,k,j_1,k_1,p_1) \wedge \left( \Big( \sum_{j_2,k_2,p_2} bl(j,k,j_2,k_2,p_2) \Big) = c \right) \right)$$

Note that for the definition of $ha(j,k)$, it is important that $HP_j$ also includes tasks with equal priority than $j$. This is because, in the RTOS scheduling policy we consider, tasks can only preempt tasks with strictly lower priority. In Figure 2, $ha(j,k) = 0$ for all task executions $(j,k)$, since in no case there are 2 tasks active when a task is waiting.

**Dependent active tasks.** Let $da(j,k)$ be the number of time quanta in the interval $\big[at(j,k), st(j,k)\big)$ where task executions depending on $j$ is active. Consider the summation indexes $j_1$, $k_1$, $p_1$ respectively defined in the sets $DS_j$, $K_{j_1}$, and $P_{j_1}$:

$$da(j,k) \stackrel{\text{def}}{=} \sum_{j_1,k_1,p_1} bl(j,k,j_1,k_1,p_1)$$

In Figure 2, $da(j_2,1) = 1$, because $j_1$ is active for the time quantum $t = 5$ between the arrival and the start of $j_2$.

**Dependent preempted tasks.** Let $dp(j,k)$ be the number of time quanta in the interval $\big[at(j,k), st(j,k)\big)$ where task executions depending on $j$ have been preempted. Consider the summation indexes $j_1$, $k_1$, $p_1$ respectively defined in the sets $DS_j$, $K_{j_1}$, and $P_{j_1}$. Then, we define

$$dp(j,k) \stackrel{\text{def}}{=} \sum_{j_1,k_1,p_1} pm(j_1,k_1,p_1) \cdot bl(j,k,j_1,k_1,p_1)$$

In Figure 2, $dp(j,k) = 0$ for all task executions $(j,k)$, since there are no dependent task preempted that block the execution of any task.

**System load.** Let $ld(t)$ be the load of the system at time $t$, i.e., the number of tasks active at time $t$. Consider the summation indexes $j$, $k$, $p$ respectively defined in the sets $J$, $K_j$, and $P_j$. Then we define

$$ld(t) \stackrel{\text{def}}{=} \sum_{j,k,p} \big(ac(j,k,p) = t\big)$$

In Figure 2, $ld(0) = 2$, and $ld(3) = 1$.

### 4.3 Constraints

We define five groups constraints related to different aspects of the RTOS.

**Well-formedness Constraints**, specifying relations among variables that directly follow from their definition in the schedulability theory.

*Each task execution starts after its arrival time, and ends after the task duration.*

$$\forall j \in J,\ k \in K_j\ \cdot\ at(j,k) \leq st(j,k) \leq end(j,k) - dr(j) \qquad \text{(WF1)}$$

*Arrival times of aperiodic tasks are separated by their minimum and maximum inter-arrival times.*

$$\forall j \in J_a,\ k \in K_j \setminus \{0\}\ \cdot\ at(j,k-1) + mn(j) \leq at(j,k) \leq at(j,k-1) + mx(j) \qquad \text{(WF2)}$$

*The time indexes $p \in P_j$ define an order over the active time quanta of tasks.*

$$\forall j \in J,\ k \in K_j,\ p \in P_j \setminus \{0\}\ \cdot\ ac(j,k,p-1) < ac(j,k,p) \qquad \text{(WF3)}$$

**Temporal Ordering Constraints**, specifying the relative ordering of tasks basing on their dependency and triggering relations.

*Each triggered task is executed the same number of times of its triggering task.*

$$\forall j_1 \in J,\ j_2 \in TS_j\ \cdot\ te(j_1) = te(j_2) \tag{TO1}$$

*Each triggered task execution arrives when its triggering task execution ends.*

$$\forall j_1 \in J,\ k \in K_{j_1}\ j_2 \in TS_j\ \cdot\ end(j_1, k) = at(j_2, k) \tag{TO2}$$

*Executions of dependent tasks cannot overlap, i.e., one task can only start after the one it depends on has ended.*

$$\begin{aligned}\forall j_1 \in J,\ k_1 \in K_{j_1}\ j_2 \in DS_j,\ k_2 \in K_{j_2} \cdot en(j_1, k_1) \leq st(j_2, k_2) \vee \\ en(j_2, k_2) \leq st(j_1, k_1)\end{aligned} \tag{TO3}$$

*If two tasks that depend on each other arrive at the same time, the higher priority task executes first.*

$$\begin{aligned}\forall j_1 \in J,\ k_1 \in K_{j_1},\ j_2 \in \big(DS_j \cap (J \setminus HP_j)\big),\ k_2 \in K_{j_2} \cdot \\ at(j_1, k_1) = at(j_2, k_2)\ \implies\ st(j_1, k_1) < st(j_2, k_2)\end{aligned} \tag{TO4}$$

**Multi-core Constraint**, capturing the specification of the number $c$ of cores of the computing platform, and stating that no more than $c$ tasks are allowed to be active in parallel at any time.

*The system load should be less than the number of cores at any time.*

$$\forall t \in T\ \cdot\ ld(t) \leq c \tag{MC}$$

Note that, when $c = 1$, MC is equivalent to an *alldifferent* constraint over $ac$.

**Preemption Constraint**, capturing the priority-driven preemptive scheduling of the RTOS, and stating that each task should be preempted when a higher priority task is ready to be executed and no cores are available.

*The number of time quanta where a task execution is preempted times $c$ is equal to the number of time quanta where higher priority tasks are active.* Considering the summation indexes $j_1$, $k_1$, $p_1$ respectively defined in the sets $HP_j$, $K_{j_1}$, and $P_{j_1}$,

$$\begin{aligned}&\forall j \in J,\ k \in K_j,\ p \in P_j\ \cdot \\ &pm(j, k, p) \cdot c = \sum_{j_1, k_1, p_1} \big(ac(j, k, p-1) < ac(j_1, k_1, p_1) < ac(j, k, p)\big)\end{aligned} \tag{PC}$$

**Scheduling Efficiency Constraint**, ensuring that there is no unnecessary task preemption, and that tasks are executed as soon as possible.

*For each time quanta in which a task execution $(j, k)$ is waiting, there should be either (1) exactly $c$ tasks with higher priority that do not depend on $j$ active, or (2) one task execution dependent on $j$ that is active, or (3) one task execution dependent on $j$ that is preempted.*

$$\forall j \in J,\ k \in K_j\ \cdot\ wt(j, k) = ha(j, k) + da(j, k) + dp(j, k) \tag{SE}$$

### 4.4 Objective Functions

We formalized three objective functions, each modeling one performance requirement, and each meant to be maximized in a separate constraint model having the same constants, variables, and constraints. In this way, solutions to each of the three constraint models characterize worst-case scenarios for the requirement modeled by the function.
**Task Deadline Misses Function**, modeling the performance requirement involving task deadlines with the function $F_{DM}$:

$$F_{DM} = \sum_{j \in J,\ k \in K_j} 2^{\,dm(j,k)}$$

To properly reward scenarios with deadline misses, $F_{DM}$ assigns an exponential contribution to deadline misses towards the sum [11]. Recall from Section 4.2 that $dm(j,k)$ is positive if the task execution $(j,k)$ misses its deadline, and negative otherwise.
**Response Time Function**, modeling the system response time with the function $F_{RT}$:

$$F_{RT} = \left( \max_{j \in J,\ k \in K_j} en(j,k) \right) - \left( \min_{j \in J,\ k \in K_j} at(j,k) \right)$$

$F_{RT}$ measures the total length in time quanta of the schedule, starting from when the first task arrives, up to when the last ends. This function is also known in traditional scheduling as *makespan*.
**CPU Usage Function**, modeling the system CPU usage with the function $F_{CU}$:

$$F_{CU} = \frac{\sum_{t \in T} \big( ld(t) > 0 \big)}{tq}$$

$F_{CU}$ measures the average CPU usage of the system over $T$, by counting all the time quanta where at least one task is active, i.e., where the system load is greater than 0.

### 4.5 Search Heuristic

We defined a search heuristic that refines the branching process of the CP OPTIMIZER solving algorithm. The heuristic specifies that the solver should mimic the behavior of a RTOS by first trying to schedule tasks with higher priority. This is done by choosing the $ac$ variables to branch on by decreasing priority, and then by assigning their time values in increasing order. For example, consider a system where $c = 1,\ j_0, j_1 \in J,\ pr(j_1) > pr(j_0)$. Suppose that, for given $k_0, p_0, k_1, j_1$ the filtering algorithm reduced the domains of the $ac$ variables to the set $[0,1]$. Figure 3a shows the branching tree in case the solver runs with default settings.

In the root node, the $ac$ variables have domain $[0,1]$. The solver then tries the first variable assignment in the branch $b_1$, stating that $j_0$ is executing at time 0. Then, the solver tries the second assignment in the branch $b_2$, stating that $j_1$ is executing at time 0. This variable assignment violates the multi-core constraint MC since both $j_0$ and $j_1$ are executing at the same time. Therefore, the solver prunes the node, backtracks to the father node, and tries the assignment in $b_3$ where $j_1$ is executing at time 1. This assignment violates the preemptive scheduling constraint PC, since $j_1$ has higher priority, but

$j_0$ is running instead. Only after backtracking up to the root node, the solver tries the assignments in $b_4$ and $b_5$ which do not violate any constraint. Note that several other branching steps might have been necessary if $ac(j_1, k_1, p_1)$ had a larger domain.

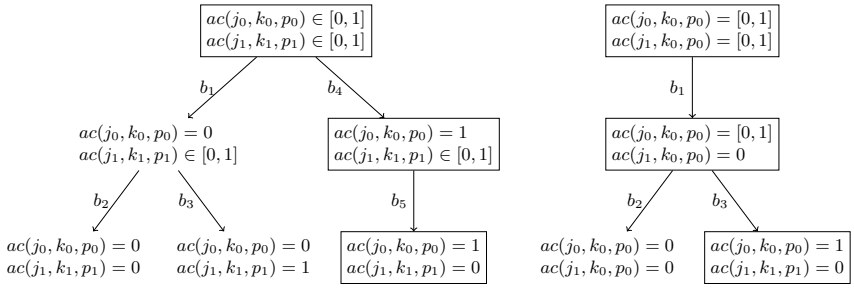



Fig. 3: Branch and bound backtracking without (a) and with (b) our search heuristic

Consider Figure 3b, where the solver has been instructed to first branch by assigning the smallest value in its domain to the $ac$ variable associated with the highest priority task. In this case, the solver tries the first assignment $ac(j_1, k_1, p_1) = 0$ in the branch $b_1$. Then, it tries the second assignment in the branch $b_2$, that violates MC. However, the third assignment in $b_3$ does not violate any constraint, making the solver perform only one backtracking step.

The semantics of this heuristic, i.e., highest priority tasks should be scheduled first, is the same as the semantics of the RTOS scheduler, which in turn is captured by preemptive scheduling constraint. By using this concept in the branching process, the solver will be less likely to assign values for $ac$ that violate the preemptive scheduling constraint, and thus will find solutions faster. We implemented the search heuristic within a stand-alone application that solves the OPL model using the .NET CONCERT library to interface with the CP OPTIMIZER. Experimentation with our search heuristic showed a significant decrease in the time needed by the solver to find solutions [11].

## 5 Industrial Experience

**Context and Process.** The work reported in this paper originates from the interaction over the years with Kongsberg Maritime (KM)[1], a leading company in the maritime and energy field. KM has pressing needs to improve its practices related to safety certification, and this involves improving the validation of performance requirements. Therefore, we proposed the work reported in this paper to provide support for systematic performance testing.

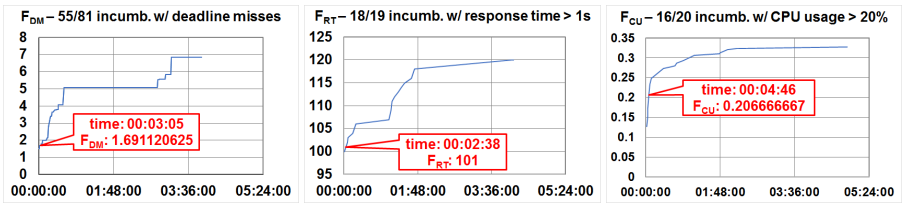
Through regular meetings with KM engineers, we first identified the need for a model-based testing approach defining the abstractions required for performance analysis [22]. Then, we casted such analysis as an optimization problem over a mathematical model of the tasks preemptive scheduling policy. To prepare for industrial adoption,

[1] `www.km.kongsberg.com`

we initially evaluated our methodology in five publicly available case studies of several RTES domains [11]. This preliminary evaluation showed encouraging results when comparing our approach with a state-of-the-art optimization strategy based on Genetic Algorithms [8]. In this paper, we provide an improved constraint model and evaluate it in the context of the KM Fire and gas Monitoring System.

**Results.** The main goal of our evaluation is to investigate whether CP can effectively be used for performance testing in an industrial context. For our approach to be used in practice, we need to discuss (1) whether the input data to our approach, i.e., the values for the constants in the constraint model, can be provided with reasonable effort, and (2) whether engineers can use the output of our analysis, i.e., the values for the variables in the constraint model, to derive stress test cases for different performance requirements. We discussed the first question in our previous work [22]. Specifically, we demonstrated that the effort to capture the input data for our approach was approximately 25 man-hours of effort. This was considered worthwhile as such drivers have a long lifetime and are certified regularly. We discuss the second question below.

Recall from Section 2 that we characterize stress test cases by arrival times of aperiodic tasks in the FMS drivers. Therefore, such arrival times are the main variables in our constraint model. We performed an experiment with the FMS drivers with an observation interval $T$ of five seconds, assuming time quanta of 10 ms. We run our OPL model for three times on a single Amazon EC2 m2.xlarge instance [1]. Each run maximized one objective function defined in Section 4.4, and had a duration of five hours. Figure 4 shows the feasible solutions with the best objective value that were found within five hours. Consistent with the terminology used in Integer Programming, we refer to these solutions as *incumbents* [3]. In each graph, the x-axis reports the incumbent computation times in the format *hh:mm:ss*, and the y-axis reports the corresponding objective value. The constraint problems had almost 600 variables and more than one million constraints, and used up to 10 GB RAM during resolution.




(a) $F_{DM}$ value over time (b) $F_{RT}$ value over time (c) $F_{CU}$ value over time

Fig. 4: Objective values over time, where we highlighted the time when the first incumbent predicted to violate a performance requirement was found

Note that, for practical use, software testing has to accommodate time and budget constraints. It is then essential to investigate the trade-off between the time needed to generate test cases, and their revealing power for violations of performance requirements. For this reason, we also recorded the computation times of the first incumbents predicted to violate the three performance requirements as expressed in Section 2.

[1] `http://aws.amazon.com`

The run optimizing $F_{DM}$ is shown in Figure 4a. The solver found 55 out of a total of 81 incumbents with at least one deadline miss in their schedule; the first of such solutions was found after three minutes. The solution yielding the best value for $F_{DM}$ produced a schedule where the *PushData* task missed its deadline by 10 ms in three executions over $T$. Figure 4b shows the results for the run optimizing $F_{RT}$. The solver found 18 out of 19 incumbents with response time higher than one second; the first of such solutions was found after two minutes. The best solution with respect to $F_{RT}$ produced a schedule where the response time of the system was 1.2 seconds. Finally, the solutions found by optimizing $F_{CU}$ are shown in Figure 4c. The solver found 16 out of 20 incumbents with CPU usage above 20%; the first of such solutions was found after four minutes. The solution with the highest value for $F_{CU}$ produced a schedule where the CPU usage of the system was 32%. In all of the three runs the solver terminated after five hours, our time budget, without completing the search with proof of optimality. However, for each objective function, the solver was able to find, within few minutes, solutions that are candidates to stress test the system as they may lead to requirements violations. These solutions can be used to start testing the system while the search continues, because the highest the objective value, the more likely the solutions are to push the system to violating its performance requirements.

## 6 Conclusions and Future Work

Currently, KM engineers spend several days simulating the behavior of the FMS and monitoring its performance requirements. We expect that, by following the systematic strategy proposed in this paper, they can effectively derive stress test cases to produce satisfactory evidence that no safety risks are posed by violating performance requirements at runtime. We note that our methodology draws on context factors (Section 4) that need to be ascertained prior to successful application. While the generalizability of these factors needs to be further studied, we have found the factors to be commonplace in many industry sectors relying on RTES. Furthermore, we note how casting the worst-case scenario analysis as a search problem relies on modeling the property to stress test as an objective function to be maximized. This is a flexible design when it comes to adapting the constraint model to test different performance requirements. In such cases, it is only needed to substitute the objective function with one modeling a difference performance requirement. Moreover, the final users of our approach, i.e., software testers and engineers, do not need to be aware of the mathematical details of the constraint model, as they can simply use our methodology as a black box test cases generator. Finally, note that there is no randomization process in the search: this means that solving a model multiple times will always find the same set of solutions in a given time budget. To achieve diversity among the test cases, we plan to consider hybrid approaches combining CP with meta-heuristic search strategies. As future work, we also plan to further investigate the scheduling capabilities of CP OPTIMIZER and OPL.

**Acknowledgments.** We gratefully thank Arnaud Malapert and Jean-Charles Régin for their invaluable input in refining the constraint model. The first and fourth authors acknowledge funding from the Research Council of Norway (ModelFusion project and Certus). The second and the third authors are supported by the Luxembourg National Research Fund (FNR/P10/03 Validation and Verification Laboratory).

## References


1. Afzal, W., Torkar, R., Feldt, R.: A systematic review of search-based testing for non-functional system properties. Information and Software Technology 51(6), 957–976 (2009)
2. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on. pp. 414–425. IEEE (1990)
3. Atamtürk, A., Savelsbergh, M.W.: Integer-programming software systems. Annals of Operations Research 140(1), 67–124 (2005)
4. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-based scheduling: applying constraint programming to scheduling problems, vol. 39. Springer (2001)
5. Beizer, B.: Software testing techniques. Dreamtech Press (2002)
6. Bell, R.: Introduction to IEC 61508. In: Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55. pp. 3–12. Australian Computer Society, Inc. (2006)
7. Berndt, D.J., Watkins, A.: High volume software testing using genetic algorithms. In: System Sciences, 2005. Proceedings of the 38th Annual Hawaii International Conference on. pp. 318b–318b. IEEE (2005)
8. Briand, L.C., 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), 145–170 (2006)
9. Cambazard, H., Hladik, P.E., Déplanche, A.M., Jussien, N., Trinquet, Y.: Decomposition and learning for a hard real time task allocation problem. In: Principles and Practice of Constraint Programming–CP 2004, pp. 153–167. Springer (2004)
10. Di Alesio, S., Gotlieb, A., Nejati, S., Briand, L.: Testing deadline misses for real-time systems using constraint optimization techniques. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. pp. 764–769. IEEE (2012)
11. Di Alesio, S., Nejati, S., Briand, L., Gotlieb, A.: Stress testing of task deadlines: A constraint programming approach. In: Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on. pp. 158–167. IEEE (2013)
12. Gomaa, H.: Designing concurrent, distributed, and real-time applications with UML. In: Proceedings of the 28th International Conference on Software Engineering. pp. 1059–1060. ACM (2006)
13. Henzinger, T.A., Sifakis, J.: The embedded systems design challenge. In: FM 2006: Formal Methods, pp. 1–15. Springer (2006)
14. Hladik, P.E., Cambazard, H., Déplanche, A.M., Jussien, N.: Solving a real-time allocation problem with constraint programming. Journal of Systems and Software 81(1), 132–149 (2008)
15. Jackson, D., Thomas, M., Millett, L.I., et al.: Software for Dependable Systems: Sufficient Evidence? National Academies Press (2007)
16. Jain, R.: The art of computer systems performance analysis. John Wiley & Sons (2008)
17. Kopetz, H.: Real-time systems: design principles for distributed embedded applications. Springer (2011)
18. Laborie, P.: IBM ILOG CP Optimizer for detailed scheduling illustrated on three problems. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 148–162. Springer (2009)
19. Le Pape, C., Baptiste, P.: An experimental comparison of constraint-based algorithms for the preemptive job shop scheduling problem. In: CP97 Workshop on Industrial Constraint-Directed Scheduling. Citeseer (1997)

20. Lee, E.A., Seshia, S.A.: Introduction to embedded systems: A cyber-physical systems approach. Lee & Seshia (2011)
21. Malapert, A., Cambazard, H., Guéret, C., Jussien, N., Langevin, A., Rousseau, L.M.: An optimal constraint programming approach to the open-shop problem. INFORMS Journal on Computing 24(2), 228–244 (2012)
22. Nejati, S., Di Alesio, S., Sabetzadeh, M., Briand, L.: Modeling and analysis of CPU usage in safety-critical embedded systems to support stress testing. In: Model Driven Engineering Languages and Systems, pp. 759–775. Springer (2012)
23. Nilsson, R., Offutt, J., Mellin, J.: Test case generation for mutation-based testing of timeliness. Electronic Notes in Theoretical Computer Science 164(4), 97–114 (2006)
24. Shams, M., Krishnamurthy, D., Far, B.: A model-based approach for testing the performance of web applications. In: Proceedings of the 3rd International Workshop on Software Quality Assurance. pp. 54–61. ACM (2006)
25. Singh, A.: Identifying Malicious Code Through Reverse Engineering. Springer (2009)
26. Tindell, K., Clark, J.: Holistic schedulability analysis for distributed hard real-time systems. Microprocessing and microprogramming 40(2), 117–134 (1994)
27. Weyuker, E.J., Vokolos, F.I.: Experience with performance testing of software systems: issues, an approach, and case study. Software Engineering, IEEE Transactions on 26(12), 1147–1156 (2000)
28. Yun, Y.S., Gen, M.: Advanced scheduling problem using constraint programming techniques in SCM environment. Computers & Industrial Engineering 43(1), 213–229 (2002)