Stress Testing of Task Deadlines: A Constraint Programming Approach

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*—Safety-critical Real Time Embedded Systems (RTESs) are usually subject to strict timing and performance requirements that must be satisfied for the system to be deemed safe. In this paper, we use effective search strategies whose goal is finding worst case scenarios with respect to deadline misses. Such scenarios can in turn be used to test the target RTES and ensure that it satisfies its timing requirements even under worst case conditions. Specifically, we develop an approach based on Constraint Programming (CP) to automate the generation of test cases that reveal, or are likely to, task deadline misses. We evaluate it through a comparison with a state-of-the-art approach based on Genetic Algorithms (GA). In particular, we compare CP and GA in five case studies for efficiency, effectiveness, and scalability. Our experimental results show that, on the largest and more complex case studies, CP performs significantly better than GA. Furthermore, CP offers some advantages over GA, such as it guarantees a complete search when there is sufficient time, and, being deterministic, it doesn't rely on parameters that potentially have a significant effect on the search and therefore need to be tuned. Hence, we conclude that our results are encouraging and suggest this is an advantageous approach for stress testing of RTESs with respect to timing constraints.**

Keywords*—*Real-Time Systems; Stress Testing; Constraint Programming

I. Introduction

Domains such as avionics, automotive and aerospace feature safety-critical systems, whose failure could result in catastrophic consequences. For this reason, the safety-related software components of these systems are usually subject to safety certification to be deemed safe for operation. Among many different aspects, software safety certification has to take into account performance requirements specifying constraints on how the system should react to its environment, and how it should execute on its hardware platform [1]. Specifically, widely used safety standards like IEC 61508 and IEC 26262 clearly state the importance of performance analysis for high Safety Integration Levels. 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 [2]. The concurrent nature of embedded software also entails that the order of external events triggering the systems tasks is often unpredictable [3]. Such increasing software complexity renders performance analysis and testing increasingly challenging. This aspect is reflected by the fact that most existing testing approaches target system functionality rather than performance, though the degradation in performance can have more severe consequences than incorrect system responses [4].

In this paper, we focus on a common [5] class of performance requirements concerned with tasks that should complete before a *deadline*. To satisfy these requirements, it is crucial to investigate to which extent some tasks are likely to miss their deadlines during operation. Design analysis techniques can be used for early verification of performance requirements in order to mitigate the impact of architectural changes in the software systems. For this purpose, specific methods for design-time performance analysis have been proposed [3]. Based on estimates for tasks execution times, these methods usually estimate the schedulability of a set of tasks through formulas and theorems from Real-Time Schedulability Theory [6]. However, such approaches are often too conservative, and as a result the predicted worst-case scenarios may never happen in practice. Moreover, extending these theories to multi-core processors has shown to be a challenge [7]. Another class of methods used for real-time performance analysis includes model checking approaches based on state machine models augmented with timing information. Real-time model checkers are used to prove reachability properties over such models in a way that paths leading to certain states represent scenarios in which deadline are either missed or met [8].

As opposed to early design verification techniques, our performance analysis approach [9] is more targeted at software testing. Specifically, our goal is to identify scenarios that exercise a system in a way that tasks are pushed as close as possible to their deadlines, possibly missing them. Consistently with the widely accepted definition [10], we refer to this activity as *stress testing*. The goal of the strategy we propose is finding combinations of system inputs that maximize the likelihood of task deadline misses. We characterize an input combination by a sequence of arrival times for aperiodic tasks in the target software system, and refer to it as *stress test case*. Finding such test cases is not trivial, since the set of all possible arrival times for aperiodic tasks quickly grows as the system size increases. For this reason, there is the need of search strategies that would effectively find stress test cases with high chances of deadline misses. In such cases, performance requirements are usually formalized with a mathematical function that drives the search towards optimal solutions. The most recent contribution in this direction that has been proposed for automated stress test cases generation is based on meta-heuristics and incomplete search, namely Genetic Algorithms (GA) [11].

The goal of this paper is to present and compare an alternative approach based on Constraint Programming to the problem presented above. 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 deadline misses. The choice of CP as an alternative to GA has been motivated by two main factors. While GA is an incomplete and randomized search approach that explores only part of the input space, CP performs a complete and deterministic search that ensures to find the global optimum upon termination. The second motivation for choosing CP is the fact that, unlike GA, CP is deterministic and doesn't rely on a set of parameters that have a significant impact on the search and therefore need to be tuned, such as GA crossover and mutation probabilities, population size and replacement strategy. Furthermore, CP is very well supported by both free and commercial tools that also provide APIs in several programming languages for building and developing domain-specific tools [12]. This last point is essential to develop and test any automated approach that aims at being used on industrial scale.

Contributions of this Paper. We present an approach based on Constraint Programming to automate the generation of stress test cases, and systematically evaluate it through a comparison with a state-of-the-art approach based on Genetic Algorithms. Specifically, this paper makes the following contributions:

1) We propose a tool-supported, efficient and effective approach based on CP to generate stress test cases that maximize the likelihood of task deadline misses.
2) We analyze the performance of this CP approach and compare it to a proposed GA alternative through a series of experiments on five industrial case studies. Our experimental results show that, as the size of the case studies increases, CP performs significantly better than GA in terms of quality of test cases and time required for finding them.

Structure of the Paper. The rest of the paper is organized as follows. Section II discusses the related work for analyzing timing properties in RTES, while Section III describes the problem of investigating deadline misses among concurrent tasks. Section IV presents our modeling framework and Section V details how Constraint Programming can be used to support stress testing of task deadlines. Finally, Section VI details the experiment set-up and discusses the results, while Section VII concludes the paper by summarizing the experimental results and providing some insights on potential future works.

II. Related Work

Testing multi-threaded concurrent software has largely focused on functional properties rather than on performance requirements [4]. In RTESs, performance properties have mostly been analyzed by verification approaches, such as Schedulability Theory [6] and Model Checking [8], rather than testing. The field of Performance Engineering extensively relies on profiling and benchmarking tools to dynamically analyze performance properties [13]. Such tools, however, are limited to producing a small number of system executions and require manual inspection of those executions. These tools are useful for checking the overall sanity of the system performance, but cannot replace systematic stress and performance testing.

Over the years, there has been a growing interest to use model-based approaches to performance testing, especially in the domains of distributed systems [14], [15]. In our prior work [9] we proposed a model-based approach to analyze CPU usage properties. We provided guidelines to extract the required information from models and formulated such analysis as a constraint optimization problem. This paper builds upon our earlier work and, in addition to being focused on generating stress test cases to break task deadlines, improves it in the following respects.

1) We have remarkably improved our constraint model to be able to capture reasonably large case studies. Our earlier work could handle systems with less than a handful of tasks only, whereas in this paper, we were able to analyze case studies with more than 30 tasks.
2) We have devised and implemented heuristics to significantly speed up the search process.
3) We evaluated our work by systematically comparing it with a state-of-the-art Genetic Algorithm (GA) search strategy [11]. To the best of our knowledge, there has been no prior empirical experiment comparing CP-based and GA-based approaches to stress test case generation.

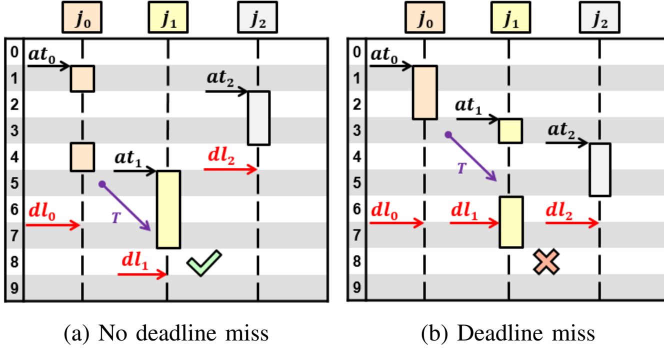
Search-based approaches like GA have previously been used to support performance testing, especially with respect to QoS constraints [14] or computational resources consumption [16]. GA have been successfully used to generate test cases for testing timeliness properties with respect to deadline misses, showing to be able to run in large systems where Model Checking approaches weren't able to run [17]. As for testing hard real-time properties such as deadline misses, the state-of-the-art is represented by the work of Briand et al. [11].

CP approaches have been used for schedulability analysis of real-time tasks subject to deadline constraints such as job-shop scheduling [18]. However, these approaches are not targeted to test case generation, such as the generation of worst case scenarios, and do not handle the specific complexities of RETSs such as multi-core architectures and preemptive scheduling policies. Nonetheless, these approaches have inspired us to consider CP in our previous [19] and current work.

III. Problem Description

Real-Time Embedded Systems (RTESs) are becoming increasingly more complex and critical in many industry sectors. A main aspect of such complexity is their concurrent architecture that entails that several tasks are triggered and executed in parallel in ways which are difficult to establish *a priori* [3]. Moreover, RTESs are often safety critical [2], and thus bound to meet strict performance requirements. In addition, their tasks must satisfy execution constraints in terms of dependency from shared computational resources, triggering of other tasks, maximum completion time, and execution priority. Given such complexity, any manual reasoning on RTES properties is very inefficient, if not infeasible.

Let us consider the system in Figure 1a featuring three tasks, j_0, j_1, j_2, in increasing priority order and executing once on a single core platform. j_0 and j_2 are aperiodic, and there is a dependency between j_0 and j_1. More specifically, j_0 triggers j_1, i.e., j_1 runs upon completion of j_0. On the contrary, the arrival time at_0 of j_0 and the arrival time at_2 of j_2 are independent.



(a) No deadline miss (b) Deadline miss

Figure 1: Impact of changes in the arrival times of tasks with respect to deadline miss properties

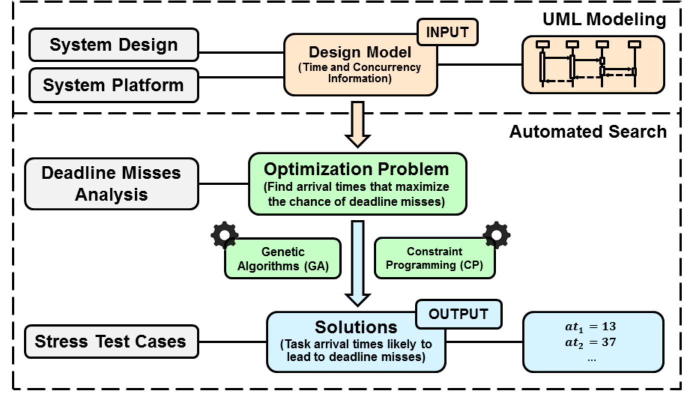


Figure 2: Our approach for schedulability risk analysis in RTES

Figures 1a and 1b represent two different execution scenarios of j_0, j_1 and j_2 corresponding to two different values for at_2, the arrival time of j_2. In the first scenario in Figure 1a, at_2 occurs before completion of j_0. Since j_2 has the highest priority, it preempts j_0 upon its arrival at at_2. Once j_2 finishes, j_0 resumes and triggers j_1 after its completion. Finally, j_1 runs. In this scenario, all the tasks j_0, j_1 and j_2 manage to finish without exceeding their deadlines. That is, all of them meet their deadline requirements. In contrast, consider the scenario in Figure 1b where at_2 occurs after completion of j_0 and while j_1 is executing. Since j_2 has the highest priority, it preempts j_1. As shown in the figure, this leads to j_1 missing its deadline.

As the above example shows, the arrival times of the tasks have a great impact on hard real-time properties, and specifically, on deadline constraints. The arrival times of the independent tasks depend on the environment and can never be predicted prior to the execution of the system. The arrival times may even vary across different system executions. In order to evaluate deadline miss constraints, we need a strategy to search all the possible task arrival times. The search has to be performed in an effective way with the objective of finding scenarios that break deadline constraints or are close to breaking them. In our work, we define a *stress test case* as a sequence of arrival times for aperiodic tasks that the search identifies as likely to lead to deadline misses. We capture the structure of the system, i.e., the tasks, their properties and their dependencies, as a constraint model. We use a constraint solver to search for the arrival time sequences that are likely to lead to deadline misses. To efficiently and effectively drive the search towards optimal solutions, we have implemented a search heuristic (Section V-B).

IV. Approach Overview

At a high level, the approach presented here is an adaptation of our earlier work [9] for deriving test cases exercising the CPU usage requirements of a RTES running on a multi-core platform. The approach has been adapted to derive test cases pushing the systems tasks as close as possible to their deadlines, and it is depicted in Figure 2. The framework blends UML modeling to capture the system design and platform, and automated search to compute stress test cases.

First, the system design and platform are modeled through sequence diagrams extended with a subset of the UML/MARTE profile capturing time and concurrency information extracted from the system specification. The profile features abstractions of the computing platform (e.g., the system scheduler with the scheduling policy and the processing unit) and the software application (e.g., tasks with their priorities, periods, dependencies, and so on). Such abstractions are needed to enable our deadline miss analysis, and will be introduced in Section V. For their mapping to the UML/MARTE profile we refer the reader to our earlier work [9] as this is not the focus of this paper.

The analysis of deadline misses is cast as an optimization problem over the abstractions represented in the design model, that therefore represents the input data for the problem. Specifically, the goal of the optimization problem is finding arrival times for aperiodic tasks that maximize the likelihood of system tasks missing their deadlines. To solve this optimization problem, we propose here an approach based on Constraint Programming (CP). Each solution of this optimization problem characterizes a test case that can be used to stress the system, i.e., to delay the completion of its tasks as much as possible, potentially missing deadlines.

V. Deadline Misses Analysis

In order to describe our deadline miss analysis, we first define the necessary abstractions in Section V-A and then present our CP approach in Section V-B.

A. Timing and Concurrency Abstractions

We define the following timing and concurrency abstractions:

- **Observation Interval.** Let T be an integer interval of length tq, i.e., $T := [0, tq - 1]$, representing the time interval during which we observe the system behavior. T is an integer interval, implying that in our analysis time is discretized. We refer to each time value $t \in T$ as a *time quantum*. In Figure 1a, $T = [0, 9]$: this means that $tq = 10$, and therefore T includes 10 time quanta.
- **Computing Platform.** Let c be the number of processor cores of the computing platform.
- **Tasks.** Let J be the set of tasks of the system. Each task $j \in J$ has a set of static properties, and a set of dynamic properties. Each property is represented by an integer value. In our analysis, we model software tasks only, as we assume that the OS tasks do not depend on software tasks. We further 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 work. In Figure 1a, $J = \{j_0, j_1, j_2\}$.

- **Static Properties of Tasks.** This is the set of properties depending on the system's design:
  - $priority(j)$, the priority of task j. For example, in Figure 1a, $priority(j_0) = 0$ and $priority(j_2) = 2$.
  - $period(j)$, the period of the task j. Only defined if j is periodic.
  - $min_ia_time(j)$ and $max_ia_time(j)$, respectively the minimum and maximum inter-arrival times, i.e., the minimum and maximum time separating two arrival times of task j. Only defined if j is aperiodic since for all periodic task j, $min_ia_time(j) = max_ia_time(j) = period(j)$ holds.
  - $duration(j)$, the estimated Worst Case Execution Time (WCET) of task j. In Figure 1a, $duration(j_0) = 2$.
  - $task_executions(j)$, the number of times the task j is executed within T. We assume it to be equal to $\lfloor T/period(j) \rfloor$ for periodic tasks, and equal to $\lfloor T/min_ia_time(j) \rfloor$ for aperiodic tasks. In this way, each aperiodic task is executed the maximum number of times possible within T. This is justified by the objective of our analysis, where we want to find worst-case scenarios that are more likely to happen when tasks are executed as many times as possible. For simplicity, we define the integer set K_j of task executions for the task j as $K_j := [1,\ task_executions(j)]$. In Figures 1a and 1b each task has only one execution.
  - $deadline(j)$, the time, with respect to its arrival time, before which j should terminate for the system not to be in an error state. We assume $\forall j \in J \cdot deadline(j) \leq period(j)$. In Figure 1a, $deadline(j_0) = 6$.
- **Dynamic Properties of Tasks.** This is the set of properties that depend on the runtime behavior of the system:
  - $arrival_time_k(j)$, the time when an event notifies the scheduler that task j should be executed for the kth time. We say that j *arrives* for the kth time at time t iff $arrival_time_k(j) = t$. In Figure 1a, $arrival_time_1(j_0) = 1$. We assume for each periodic task j, we have $arrival_time_k(j) = (k-1) \cdot period(j)$. The same assumption is made by the Generalized Completion Time Theorem (GCTT) [3] to ensure that the analysis considers the worst case where all periodic tasks simultaneously arrive for the first time at the beginning of T.
  - $start_k(j)$ and $end_k(j)$, respectively the time when j starts and finishes its kth execution. In Figure 1a, $start_1(j_0) = 1$ and $end_1(j_0) = 5$.
  - $active(j, t)$, a boolean variable that has value 1 if j is running at time t, and value 0 otherwise. In Figure 1a, $active(j_0, 1) = 1$ and $active(j_0, 2) = 0$.
  - $task_deadline_k(j)$, the absolute deadline of the kth execution of j: $task_deadline_k(j) := arrival_time_k(j) + deadline(j) - 1$. In Figure 1a, $task_deadline_1(j_0) = 6$.
  - $deadline_miss_k(j)$, the amount of time by which j missed its deadline during its kth execution, i.e., $deadline_miss_k(j) := end_k(j) - task_deadline_k(j) - 1$. Negative if $end_k(j) - 1 < task_deadline_k(j)$, non-negative otherwise. In Figure 1a, $deadline_miss_1(j_0) = -2$.
- **Relationships between Tasks.** Let the following be two binary relations defined in $J \times J$:
  - $dependent(j_1, j_2)$ holds if there exists a computational resource r such that tasks 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, but one can execute only after the other has released the lock on the resource. The relation $dependent$ is defined as reflexive and symmetric.
  - $triggers(j_1, j_2)$ holds if the event triggering the task j_2 occurs when the task j_1 finishes its execution, i.e. $\forall k \in K_{j_1} \cdot arrival_time_k(j_2) = end_k(j_1)$. In Figure 1a, $triggers(j_0, j_1)$ holds. The relation $triggers$ is irreflexive and antisymmetric.
- **Performance Requirement.** As explained in Section IV, the goal of our approach is to find values for the arrival times of aperiodic tasks that maximize the likelihood of deadline misses, and hence are more likely to violate the deadline performance requirements of the system. We formalized this concept through a function of output properties whose value captures how arrival times compare in terms of their likelihood of triggering deadline misses. Such a function is referred to as *objective function* in the context of Constraint Programming, and as *fitness function* in the context of Genetic Algorithms. We first identify a set of characteristics the function should meet:
  - *No deadline miss is overshadowed.* In safety-critical real-time systems even a single deadline miss could lead the system to a fail state. Thus, a good function should not allow task executions which meet their deadline to overshadow deadline misses.
  - *The more deadline misses, the higher the value.* Intuitively, the function value should take into account the number of deadline misses among task executions. Even if a system could recover from a scenario where a task misses its deadline in a single execution, recovering from several deadline misses might be harder.
  - *The larger the deadline misses, the higher the value.* Our analysis is based on WCET estimates ($duration$) for the system tasks. Such estimates could be over-pessimistic, and our approach could compute a test case identifying a deadline miss that will not happen when actually testing the system. However, the closest to its deadline a task is in our analysis, the more likely it is to miss a deadline in a real scenario. Such concept is captured by the quantity defined as $deadline_miss$: the larger its value, the closer the task completion time to its deadline, with possibly the task missing its deadline. Hence, we expect a good function to prioritize scenarios where a larger deadline miss is identified.

Having considered the criteria above, we adopted a modified version of the function defined by Briand et al. [11]:

$$f(j) = \sum_{k \in K_j} 2^{\,deadline_miss_k(j)} \tag{1}$$

Note that f is defined for a given task j. An alternative function which instead takes into account all tasks is given by the sum of f for each task in the system:

$$F = \sum_{j \in J} f(j) \tag{2}$$

Note that the purpose of $f(j)$ is to identify deadline miss scenarios for a single critical task j, since it has larger values when large deadline misses occur in j. On the other hand, F has larger values when more deadline misses on several tasks occur, aiming at identifying scenarios stressing the whole system rather than a single task. Given that we use our case studies for the purpose of experimental evaluation and comparison, there is no clear guidance on how to choose a specific target task j for each case, and hence, we will use F in place of $f(j)$.
As observed before, $deadline_miss_k(j)$ is positive if task j misses its deadline during its kth execution, and negative otherwise. This means that large negative values stand for j ending long before its deadline. On the other hand, positive values stand for j failing to end before its deadline, thus missing it. The exponential shape of the function favors executions with large deadline misses, thus avoiding them being overshadowed by other executions.

B. Using Constraint Programming to Support Stress Testing

Constraint Programming (CP) is a programming paradigm where relations among variables are expressed in form of constraints [20]. Constraint Programming can both be used to solve satisfiability problems, and Constraint Optimization problems (COPs), where the goal is to find a solution which maximizes a given objective function. The latter is usually solved with branch and bound algorithms that, when combined with a complete search labeling heuristic over the domains of variables, compute the global optimum of a COP. Branch-and-bound algorithms usually iterate over three steps: (1–branch) divide a set of candidate solutions into two or more partitions, (2–bound) compute bounds for the value of the objective function in one set of candidate solutions, and (3–prune) possibly discard sets of candidate solutions that are shown to be sub-optimal or infeasible. The common representation of a branch and bound algorithm is a *branching tree*, since recursively applying the branch step starting from the whole search space defines a tree structure whose nodes are the candidate solutions, and whose edges are the node branches. Branch and bound algorithms are also supported by search heuristics, i.e., problem-specific techniques used to speed up the search process, for instance by specifying the selection policy for the node to branch at each iteration. Due to its completeness, the search may take time to complete and hence heuristics are used to shorten the search time and quicken the convergence towards the global optimum. Constraint problems are represented in constraint models that include constant values, variables, constraints, and in case of COPs an objective function. Solutions for such models are found by constraint solvers, which implement solving algorithms featuring constraint propagation and domain filtering, often allowing to specify user-defined search heuristics.

In this paper, we propose a constraint optimization model implemented in the Optimization Programming Language (OPL) [21] for the purpose of generating sequences of arrival times likely to lead to deadline misses. The key idea behind our work is to model the properties of the system as integer constants and variables, and to model the scheduler of the system as a set of constraints among such variables. The constraint model consists of a set of constants, a set of variables, a set of constraints and an objective function. The model is expressed with the notation presented in Section V-A, and improved with the addition of a search heuristic. An excerpt of our constraint model is presented in Listing 1.

```
1  /* Task Constants */
2  int task_deadline[J] = ...;
3  int triggers[J, J] = ...;
4  /* Task Execution Variables */
5  dvar int arrival_time[j in J, k in K[j]] in T;
6  dvar int end[j in J, k in K[j]] in T;
7  /* Objective Function */
8  maximize sum(j in J, k in K[j]) 2^deadline_miss[j, k];
9  /* I. Well-formedness constraints */
10 forall(j in J, k in K[j])
11     wfc: end[j, k] >= start[j, k] + duration[j];
12 /* II. Temporal Ordering constraints */
13 forall(j1 in J, k in K[j1], j2 in J : triggers[j1, j2]
       == 1)
14     toc: arrival_time[j2, k] = end[j1, k];
15 /* III. Multi-core Constraint */
16 forall(t in T)
17     mcc: count(all(j in J)active[j, t], 1) <= c;
18 /* IV. Preemptive Scheduling Constraints */
19 /* V. CPU Usage Constraints. */
```

Listing 1: Excerpt of our constraint model

The constraint model defines 29 constants, 9 variables and 19 constraints. Each of those is instantiated during the solving process for each task execution. Due to lack of space, we explicitly report and discuss only some representative constants, variables and constraints. Full details about the constraint model are available as a technical report [22].

- **Constants.** Constants in the model correspond to the Static Properties of Tasks, plus details about the computing platform and the observation interval. Values for the constants are based on data defined in an external file, and are the input values of the constraint model.
- **Variables.** Variables in the model correspond to the dynamic properties of tasks, plus some technical variables used to simplify the description of the constraints. Values for the variables are computed by the constraint solver, and are the output values of the constraint model.
- **Constraints.** Constraints in the model specify mathematical relations between variables and constants, and are divided into five major subsets:
  1) **Well-Formedness Constraints.** These constraints specify relations among variables directly following from their definition. For example, the constraint *wfc* at line 11 of Listing 1 states that the end time for each task is greater (or equal in case the task is not preempted) to its start time plus its duration.
  2) **Temporal Ordering Constraints.** These constraints capture the $dependent$ and $triggers$ relationships between tasks. For example, the constraint *toc* at line 14 of Listing 1 states that the arrival time of a task $j2$ triggered by a task $j1$ is equal to the end time of $j1$.
  3) **Multi-core Constraint.** This constraint captures the specification of the number c of cores of the computing platform. For example, the constraint *mcc* at line 17 of Listing 1 specifies that for each time quantum t, no more than c tasks can be active.
  4) **Preemptive Scheduling Constraints.** These constraints capture the priority-driven preemptive behavior of the system scheduler, stating that each task should be preempted when a higher priority task is ready to be executed and no cores are available.

5) **CPU Usage Constraints.** These constraints ensure the scheduler avoids unnecessary context switching and executes tasks as soon as enough resources are available.

- **Objective Function.** The objective function of the constraint model is the one defined in Equation (2).
- **Search Heuristic.** The constraint solver used to solve the model is augmented with a search heuristic to refine the branching process of the solving algorithm. The heuristic specifies that the solver should first try to schedule tasks with higher priority by choosing the $active$ variables to branch on by decreasing priority, and then by trying to assign them time values in increasing order. For example, consider a system where $c = 1$, $j_0, j_1 \in J$, $priority(j_1) > priority(j_0)$. Figure 3a shows the branching tree in case the solver runs with the default settings. In the root node, no decision is made and the $active$ variable is assigned value 0 for each task and each time quantum. The solver then tries the first variable assignment in the branch b_1. In the first assignment, j_0 is executing at time 0, and j_1 is not. This variable assignment violates the preemptive scheduling constraint, since j_1 has higher priority. The solver then prunes the node, backtracks to the root node, and tries the assignment in b_2, which violates the multi-core constraint since both j_0 and j_1 are executing at the same time. Only after a second backtracking, the solver tries the assignment in b_3 which does not violate any constraint. Consider now Figure 3b. The solver has been instructed to first try to branch by assigning value 1 to the task with the highest priority. In this case, the first branch b_1 leads to a variable assignment not violating the constraints, and no backtracking is needed.

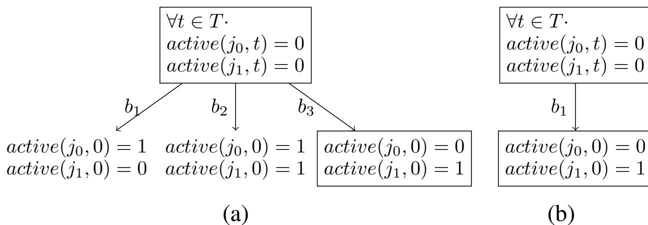


Figure 3: Branch and bound backtracking without (a) and with (b) search heuristics

It is important to note that the semantics of this heuristic (highest priority tasks should be scheduled first) is the same as the semantic of the preemptive scheduling constraint. By using this concept in the branching process, the solver will be less likely to assign values for $active$ 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 constraint model using IBM ILOG CP OPTIMIZER, one of the leading CP solvers in the market.

VI. EMPIRICAL STUDY

The goal of our empirical study is to compare the overall performance of GA and CP for the purpose of supporting stress testing of task deadlines. Recall from Section II that an approach based on GA [11] was proposed to support stress testing of task deadlines by searching for worst-case scenarios, and is therefore a natural comparison baseline. To successfully enable our empirical study, we slightly modified that GA approach. Specifically, (1) we added the support for multi-core platforms, as the original work was meant for analyzing only software systems running on single-core architectures, and (2) we replaced the fitness function in Equation (1) with the one in Equation (2), for the reasons presented in Section V-A.

The comparison is performed on five case studies reported in the literature, fully described in Section VI-A. The goal of our study is answering the research questions presented in Section VI-B based on the metrics and attributes detailed in Section VI-C. The design of our experiment is described in Section VI-D, and its results are discussed in Section VI-E. Finally, Section VI-F covers some potential threats that could affect the general validity of our conclusions.

A. Case Studies

To investigate the general performance of GA and CP in a variety of conditions, we selected five case studies from safety-critical domains with varying size and complexity. Specifically, our comparison is based on one case study from the aerospace domain, two case studies from the automotive domain, and two from the avionics domain. The systems presented in the following case studies share the most common characteristics of safety-critical RTES: they are integrated with the physical domain by interacting with external devices such as sensors and actuators, they have a concurrent design, and they are subject to timing requirements ranging in the order of milliseconds.

- **Ignition Control System (ICS).** Bosch GmbH developed an ignition control system of an automotive engine [23]. The system features sensors and actuators to sample physical phenomena such as knock, temperature variation and engine warm-up, and to perform corrections over them for a successful ignition of a spark plug in the engine.
- **Cruise Control System (CCS).** Continental AG developed a Cruise Control System deployed on AUTOSAR-compliant architectures [24]. The system features a switch sensor that acquires driver inputs (e.g., set/cancel cruise, increase/decrease speed), and a control system that processes the inputs and maintains the specified vehicle speed.
- **Unmanned Air Vehicle (UAV).** The ENSMA together with the University of Poitiers in France worked on a joint project for a mini Unmanned Air Vehicle named AMADO [25]. The system embeds a camera to be able to follow dynamically defined way-points, and is connected to a ground station via a wireless modem that allows it to receive instruction data during a mission.
- **Generic Avionics Platform (GAP).** The Software Engineering Institute, the Naval Weapons Center and IBM's Federal Sector Division designed a specification for a hypothetical avionics software mission control computer of a military aircraft [26]. Though the system can be configured to fit several possible missions, the specification is targeted for the specific case of an air-to-surface attack.
- **Herschel-Planck Satellite System (HPSS).** The European Space Agency carried out the Herschel-Planck Mission consisting the two satellites Herschel and Planck [27]. The satellites have different scientific purposes: Herschel carries a large infrared telescope, while Planck is a space observatory for studying the Cosmic Microwave

Background. The satellites share the same computational architecture composed of a real-time operating system, a basic software layer, and application software.

Table I summarizes relevant data from the case studies specifications, reported in ascending order of size and complexity. Specifically, we take into account the number of software tasks, inter-dependencies, triggering relations, and platform cores.

	Software System				Platform
	Tasks		Relationships		Cores
	Periodic	Aperiodic	Dependencies	Triggering	
ICS	3	3	3	0	3
CCS	8	3	3	6	2
UAV	12	4	4	0	3
GAP	15	8	6	5	2
HPSS	23	9	5	0	1

Table I: Case studies data

B. Research Questions

The goal of our empirical study is answering the following research questions involving GA and CP for the purpose of supporting stress testing of task deadlines.

- **RQ1 — Efficiency**. Does one search technology find the best solutions significantly faster than the other?
- **RQ2 — Effectiveness**. Does one search technology find significantly better solutions (i.e., solutions with worse deadline misses) than the other?
- **RQ3 — Scalability**. To what extent does the size of a system affect the efficiency and effectiveness of the two search technologies?

RQ1 and RQ2 are investigated through a set of metrics and attributes detailed in Section VI-C. The goal of such metrics and attributes is to provide quantitative evidence to answer the research questions. RQ3 will instead be only qualitatively discussed in Section VI-E. This is because we base our analysis of efficiency and effectiveness on a set of five case studies, and therefore no quantitative analysis, for example based on regression analysis, can be carried out.

C. Comparison Metrics and Attributes

Though the search for optimal solutions is driven by function F defined in Equation 2, we broke down F into several factors that are of practical interest while investigating worst case scenarios for deadline misses. This is because, to properly answer the research questions, one must look into several complementary aspects of F. For this reason, we defined the efficiency and effectiveness properties related to RQ1 and RQ2 as *attributes*, and we defined a set of *metrics* to enable their measurement. Therefore, we compare the performance of GA and CP by collecting data pertaining to the metrics and attributes defined below.

The following metrics are defined for a given solution x found by the search technology $\Gamma \in \{GA, CP\}$ during an experiment on the target system $\Sigma \in \{ICS, CCS, UAV, GAP, HPSS\}$.

- **Computation time $\mathbf{t}$**. We define $t(x)$ as the time required to find solution x, from when the search starts.
- **Sum $\mathbf{s}$ of time quanta in deadline misses**. We define $s(x)$ as the sum of time quanta in all deadline misses of solution x. Recall from Section V that, for a given solution x, we define $s = \sum_{j,k} max\,(0, end_k(j) - deadline_miss_k(j))$. The sum of time quanta in all deadline misses is strongly related to the value of the fitness/objective function that guides the search. In practice, the sum of time quanta in deadline misses provides some insight on the magnitude of the identified deadline misses. Since our approach is based on task execution time estimates, the larger the sum of deadline misses, the more likely tasks are to miss their deadlines at runtime.
- **Number $\mathbf{n}$ of tasks that miss a deadline**. We define $n(x)$ as the number of tasks that miss at least a deadline in solution x. This number is relevant as, in practice, every task that misses a deadline has to be looked into and possibly re-designed. Hence, not realizing a task can miss its deadline may lead to overlooking an important flaw.
- **Number $\mathbf{m}$ of task executions that miss a deadline**. We define $m(x)$ as the number of task executions that miss a deadline in solution x. This number is also of interest as, in soft real-time systems, one could tolerate less critical tasks to miss some deadlines, provided that the frequency of deadline misses is acceptable. Therefore, not estimating m correctly might lead to inspect a task when unnecessary.

We note how the metrics s, n, and m also capture the general *quality* of a solution. Intuitively, higher values for s, n and m, all correspond in a different way to higher quality solutions. In other words, solutions with many large deadlines or many tasks that miss a deadline characterize worst case scenarios. Therefore, a *best* solution can only be identified with respect to a specific metric. For each search technology Γ running during an experiment on the target system Σ, we respectively define:

- **The best solution $\mathbf{x_s^*}$ with respect to $\mathbf{s}$** as the solution that has the highest sum s^* of time quanta in deadline misses, i.e., $s(x_s^*) = s^*$.
- **The best solution $\mathbf{x_n^*}$ with respect to $\mathbf{n}$** as the solution that has the highest number n^* of tasks missing at least one deadline, i.e., $n(x_n^*) = n^*$.
- **The best solution $\mathbf{x_m^*}$ with respect to $\mathbf{m}$** as the solution that has the highest number m^* of task executions missing a deadline, i.e., $m(x_m^*) = m^*$.

The following attributes are also defined for each search technology Γ running during an experiment on the target system Σ.

- **Efficiency η**. We define the efficiency η with respect to a given metric as the time required to compute the best solution with respect to that metric. Specifically, we define the efficiency with respect to s, m, and n:
$$\eta_s = t(x_s^*) \quad \eta_n = t(x_n^*) \quad \eta_m = t(x_m^*)$$
Our definition of efficiency entails that the more efficient a technology, the faster it computes its best results with respect to some metric.
- **Effectiveness κ**. We define the effectiveness κ with respect to a given metric as the value of that metric for the best solution found. Specifically, we define effectiveness with respect to s, m, and n:

$$\kappa_s = s^* \quad \kappa_n = n^* \quad \kappa_m = m^*$$

Our definition of effectiveness entails that the more effective a technology, the higher with respect to some metric are the solutions it computes.

D. Experiments Set-Up

To answer RQ1, RQ2, and RQ3, we performed a series of experiments over the systems described in Section VI-A. The experimental design is illustrated in Figure 4. Each experiment consisted of running both search technologies on a target system for a number of times, each run having the same duration. Since the purpose of our empirical study is to compare the practical usefulness of GA and CP, we chose to run each technology in the way engineers would realistically do so in a real testing environment. Based on our experience with industrial partners, we assumed that a reasonable choice would be running both technologies on a desktop computer for one hour. To do so, we set up GA to continuously generate new solutions for one hour, while we set up CP to terminate the search after one hour. Running both search technologies for the same amount of time allows us to meaningfully compare their effectiveness. Furthermore, during the design of the experiment, we had to consider the inherent randomized behavior of GA in contrast to the fully deterministic behavior of CP. Indeed, GA finds solutions starting from a randomly chosen initial population of individuals by applying crossover and mutation operators with a given probability, while CP finds solutions by solving a constraint optimization problem. For this reason, while we ran CP only once for one hour for each system, we ran GA 50 times for one hour on each system. In this way, we could compute distributions of the best solutions recorded over 50 runs over the efficiency and effectiveness ranges. For each experiment, we recorded only the 100 solutions with the highest fitness/objective value found by GA and CP. Since each solution characterizes a stress test case, 100 is a reasonable number of stress test cases to run [28]. Since RQ1 and RQ2 are respectively related to attributes η and κ, for each solution we computed the values of the metrics t, s, n, and m used to define such attributes. Both GA and CP have been run one at a time on the same machine, i.e., a desktop computer with a 3.3 Ghz dual-core Intel Core i3 processor and 8GB RAM.

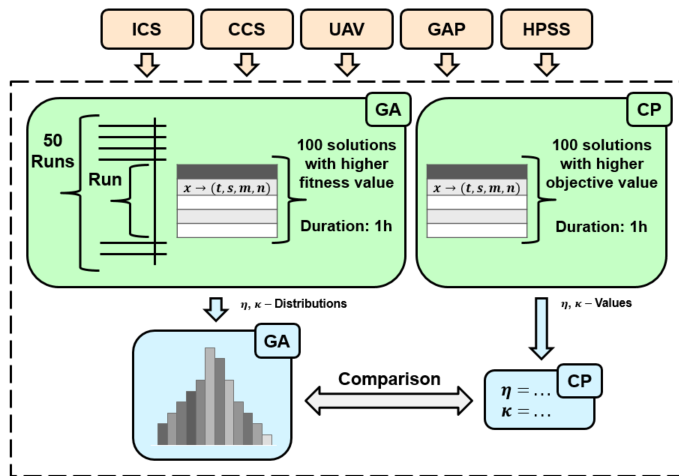


Figure 4: Experimental design

E. Results and Discussion

Table II reports efficiency η with respect to s, n and m for GA and CP and for each case study. The computation times for the best solutions are reported in the format *mm:ss*. Due to lack of space, we don't report the full distributions of GA. Instead, we report a set of statistics that meaningfully represent the efficiency of GA across runs, specifically:

- The mean computation time $\bar{x}$ of the best solution
- The three quartiles Q_1, Q_2, and Q_3 of the computation time of the best solution
- The probability P that GA will achieve a greater or equal efficiency than CP. P is calculated as the percentage of runs in which GA had a greater or equal efficiency than CP, i.e., the percentage of runs in which GA found its best result before or at the same time as CP found its own.

Being deterministic, the column of CP reports instead the single computation times of the best solutions.

	η_s			η_n			η_m		
	GA		CP	GA		CP	GA		CP
ICS	$\bar{x}$	15:23		$\bar{x}$	11:05		$\bar{x}$	11:05	
	Q_1	09:33		Q_1	04:33		Q_1	04:33	
	Q_2	14:07	40:23	Q_2	07:49	40:23	Q_2	07:49	40:23
	Q_3	18:05		Q_3	13:32		Q_3	13:32	
	P	0.98		P	1		P	1	
CCS	$\bar{x}$	24:42		$\bar{x}$	07:20		$\bar{x}$	07:20	
	Q_1	15:09		Q_1	05:19		Q_1	05:19	
	Q_2	22:33	18:04	Q_2	06:48	18:04	Q_2	06:48	18:04
	Q_3	30:52		Q_3	08:16		Q_3	08:16	
	P	0.36		P	1		P	1	
UAV	$\bar{x}$	42:01		$\bar{x}$	39:50		$\bar{x}$	39:50	
	Q_1	33:39		Q_1	32:49		Q_1	32:49	
	Q_2	38:34	01:05	Q_2	37:11	00:37	Q_2	37:11	00:37
	Q_3	53:29		Q_3	48:19		Q_3	48:19	
	P	0		P	0		P	0	
GAP	$\bar{x}$	40:26		$\bar{x}$	21:07		$\bar{x}$	30:03	
	Q_1	33:00		Q_1	06:30		Q_1	10:59	
	Q_2	40:32	22:38	Q_2	12:47	01:38	Q_2	34:50	01:38
	Q_3	50:22		Q_3	34:20		Q_3	42:48	
	P	0.1		P	0		P	0	
HPSS	$\bar{x}$	20:19		$\bar{x}$	20:19		$\bar{x}$	20:19	
	Q_1	14:31		Q_1	14:31		Q_1	14:31	
	Q_2	17:51	05:56	Q_2	17:51	00:54	Q_2	17:51	00:54
	Q_3	22:30		Q_3	22:30		Q_3	22:30	
	P	0		P	0		P	0	

Table II: Experiments results for efficiency η

We observe how, on the two smallest case studies, GA has a consistently better efficiency than CP. Specifically, in ICS, GA was able to find on average the best solutions x_s^*, x_n^*, and x_m^* three or four times faster than CP. We can identify this trend also in CCS, where we recorded the same efficiency gap with the exception of η_s, where the efficiency of CP is achieved by GA by the second quartile. However, for the three largest case studies, CP is significantly faster than GA at finding the best results with respect to s, n, and m. The efficiency of CP is indeed far above the one observed before the third quartile of GA. With the exception of η_s in GAP, no GA run was faster at finding its best result than CP.

Table III reports the effectiveness κ with respect to s, n, and m for GA and CP and for each case study. As for Table II, the columns of GA report statistics about the distribution of effectiveness:

- The mean value $\bar{x}$ of the best solution
- The three quartiles Q_1, Q_2, and Q_3 of the value of the best solution

- The probability P that GA will achieve a greater or equal effectiveness than CP. P is calculated as the percentage of runs in which GA had a greater or equal effectiveness than CP, i.e., the percentage of runs in which the best result found by GA was better than or equal to the best result found by CP.

The column of CP reports instead the single value of the best solutions.

	κ_s			κ_n			κ_m		
	GA		CP	GA		CP	GA		CP
ICS	$\bar{x}$	13.22		$\bar{x}$	1.3		$\bar{x}$	1.3	
	Q_1	14		Q_1	1		Q_1	1	
	Q_2	14	19	Q_2	1	2	Q_2	1	2
	Q_3	19		Q_3	2		Q_3	2	
	P	0.26		P	0.32		P	0.32	
CCS	$\bar{x}$	12.14		$\bar{x}$	2		$\bar{x}$	2	
	Q_1	11		Q_1	2		Q_1	2	
	Q_2	13	13	Q_2	2	2	Q_2	2	2
	Q_3	13		Q_3	2		Q_3	2	
	P	0.52		P	1		P	1	
UAV	$\bar{x}$	0.94		$\bar{x}$	0.74		$\bar{x}$	0.74	
	Q_1	0		Q_1	0		Q_1	0	
	Q_2	1	3	Q_2	1	1	Q_2	1	1
	Q_3	1		Q_3	1		Q_3	1	
	P	0.02		P	0.74		P	0.74	
GAP	$\bar{x}$	19.18		$\bar{x}$	2.4		$\bar{x}$	3.06	
	Q_1	16		Q_1	2		Q_1	3	
	Q_2	19	34	Q_2	2	3	Q_2	3	5
	Q_3	21		Q_3	3		Q_3	4	
	P	0		P	0.4		P	0.02	
HPSS	$\bar{x}$	0.04		$\bar{x}$	0.04		$\bar{x}$	0.04	
	Q_1	0		Q_1	0		Q_1	0	
	Q_2	0	5	Q_2	0	1	Q_2	0	1
	Q_3	0		Q_3	0		Q_3	0	
	P	0		P	0.04		P	0.04	

Table III: Experiments results for effectiveness κ

We observe how, on the two smallest case studies, the effectiveness of GA is on average similar to the effectiveness of CP. In ICS, GA reaches by the third quartile the same result as CP for κ_s, κ_n, and κ_m. In CSS instead, GA reaches by the second quartile the same result as CP for κ_s, and does so by the first quartile for both κ_n and κ_m. However in both cases, though the solutions found by CP are better on average, the efficiency of GA is superior to the efficiency of CP. This means that there is a high probability that in few runs GA will find the same best solutions with respect to s, n, and m as CP. For the three largest case studies instead, with the exception of UAV for κ_n and κ_m, CP finds significantly better values than GA for s, n, and m. Specifically, in UAV GA finds on average one deadline miss of one time quantum, while CP finds one deadline miss of three time quanta. The difference in κ_s between GA and CP increases in GAP and HPSS. In GAP, GA has an average value of 19 for κ_s, while CP achieves 34 half of the time. In HPSS, GA hardly finds any deadline miss, while CP finds one of five time quanta after a few minutes. These differences in the value of κ_s are of practical significance because, as discussed above, a larger sum of deadline misses indicates scenarios where tasks are more likely to miss their deadlines at runtime. Furthermore, for all case studies, no GA run found a better solution than CP.

In light of these results, we conclude that, for the smaller case studies, GA has proven to be more efficient than CP, and nearly as effective. On the other hand, for the larger case studies, CP has proven to be significantly more efficient and more effective than GA. Our results show that, the larger the size of the system, the better CP when compared to GA within the range covered by our case studies.

We conjecture that the reason for this trend stems from the interaction between the size of the search space of the case studies and the heuristics used in our CP solution. The size of the search space is largely determined by the number of aperiodic tasks executions within the observation time interval. The case studies where GA is more efficient than CP, i.e., ICS and CCS, have a smaller search space, and hence, GA is able to quickly converge towards the best solutions regardless of its initial population. Therefore, without being augmented with any heuristics, GA performs reasonably well compared to CP. On the other hand, in case studies with large search spaces, i.e., UAV, GAP, HPSS, GA needs more time to converge towards its best solution. Thanks to its heuristic, CP is able to find solutions with a rather high objective value in a few minutes. Note that the existing efficiency and effectiveness results of CP are obtained by running it for one hour. Due to this time limit, in our larger case studies, CP cannot further improve the solutions that it finds at the very beginning of its search. If CP had been run for longer time on the larger case studies, we might have obtained solutions that take much longer to compute and have higher objective values. Finally, we note that for case studies with much larger search spaces than our existing case studies, the CP solution may fail due to not being able to construct the constraint model in memory.

F. Threats to Validity

We identified three main threats that could affect the general validity of our conclusions: First, the analysis of efficiency, effectiveness and scalability is based on a set of five case studies. Although comparing GA and CP in a larger number of systems would have mitigated this threat, the case studies have been selected from different RTES domains and feature varying size and complexity.

Second, the size of the selected case studies varies from 6 to 32 tasks, 3 to 9 of which aperiodic. There could be much larger case studies featuring hundreds of tasks, and for those the efficiency and effectiveness of CP need to be investigated. This means that the conclusions drawn are valid only for systems in the same size range of the case studies used in the comparison.

Third, the experiment set-up relies on the choice of running both technologies for one hour, and on specific parameters used for GA, such as the initial population size, the crossover and mutation probabilities, and the population replacement rate. Different values for these parameters could have led to higher efficiency and effectiveness. However, we used the same values as in the work of Briand et al. [11], that have been derived from the GA literature and specifically tuned for deadline misses analysis. Moreover, by looking at the quartiles of the efficiency distributions, GA found its best results significantly earlier than 1 hour. This means that in most cases, GA reached a plateau before 1 hour, and chances for it to find a better solution if given more time are likely to be low.

VII. Conclusions and Future Work

In this paper, we propose a novel Constraint Programming (CP) approach to support stress testing of task deadlines by identifying worst case scenarios where tasks are more likely to miss such deadlines. The approach is proposed as an alternative to the state-of-the-art relying on metaheuristic search, such as

Genetic Algorithms (GA). CP offers a number of potential advantages over GA, which makes its investigation worthwhile in our context: it can potentially guarantee the completeness of the search provided it has sufficient time, and, being deterministic, it doesn't need its users to take into account parameters such as mutation and crossover probabilities, population size and replacement rate.

The proposed approach expresses deadline misses analysis as a Constraint Optimization Problem (COP). The COP is implemented in a constraint model defined in OPL, and solved via the IBM ILOG CP OPTIMIZER in a stand-alone tool. The solutions found by CP can be used to characterize stress test cases that are crucial for building satisfactory evidence to demonstrate that no safety risks are posed by potential task deadline misses. We define the concepts of efficiency and effectiveness as means to evaluate the overall performance of our approach in terms of time required to compute the best solution and the quality of the best solution computed. We empirically investigate the efficiency and effectiveness of our approach by performing a series of experiments over five case studies that compare CP to GA. The results show that, for the larger and more complex case studies, CP achieves significantly better efficiency and effectiveness than GA, within the size range covered. Finally, we note that, similarly to GA, CP can always provide results within a time budget, even when not terminating with proof of optimality.

Our approach relies on a number of context factors (Section V) which need to be ascertained before CP can be applied in industry for supporting stress testing of task deadlines. Although the generalizability of these factors needs to be further studied, our experience suggests that these factors are commonplace in many industrial real-time embedded systems. In the future, we plan to investigate the applicability of CP on larger case studies to better evaluate its scalability, and perform further experimentation with hybrid approaches combining complete and meta-heuristic search strategies.

Acknowledgments. We gratefully acknowledge funding from the Research Council of Norway (ModelFusion project and Certus). The second and the third authors are supported by the National Research Fund, Luxembourg (FNR/P10/03Validation and Verification Laboratory).

REFERENCES

[1] D. Jackson, M. Thomas, L. I. Millett *et al.*, *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, 2007.

[2] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Springer, 2011.

[3] H. Gomaa, "Designing concurrent, distributed, and real-time applications with UML," in *Proceedings of the 28th International Conference on Software Engineering*. ACM, 2006, pp. 1059–1060.

[4] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *Software Engineering, IEEE Transactions on*, vol. 26, no. 12, pp. 1147–1156, 2000.

[5] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[6] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocessing and microprogramming*, vol. 40, no. 2, pp. 117–134, 1994.

[7] A. David, J. Illum, K. Larsen, and A. Skou, "Model-based framework for schedulability analysis using UPPAAL 4.1," *Model-Based Design for Embedded Systems*, p. 93, 2010.

[8] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," in *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on*. IEEE, 1990, pp. 414–425.

[9] S. Nejati, S. Di Alesio, M. Sabetzadeh, and L. Briand, "Modeling and analysis of cpu usage in safety-critical embedded systems to support stress testing," in *Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 759–775.

[10] B. Beizer, *Software testing techniques*. Dreamtech Press, 2002.

[11] L. C. Briand, Y. Labiche, and M. Shousha, "Using genetic algorithms for early schedulability analysis and stress testing in real-time systems," *Genetic Programming and Evolvable Machines*, vol. 7, no. 2, pp. 145–170, 2006.

[12] F. Benhamou, N. Jussien, and B. O'Sullivan, *Trends in constraint programming*. Wiley-ISTE, 2010.

[13] R. Jain, *The art of computer systems performance analysis*. John Wiley & Sons, 2008.

[14] M. Shams, D. Krishnamurthy, and B. Far, "A model-based approach for testing the performance of web applications," in *Proceedings of the 3rd International Workshop on Software Quality Assurance*. ACM, 2006, pp. 54–61.

[15] C. Barna, M. Litoiu, and H. Ghanbari, "Model-based performance testing: NIER track," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 872–875.

[16] D. J. Berndt and A. Watkins, "High volume software testing using genetic algorithms," in *System Sciences, 2005. Proceedings of the 38th Annual Hawaii International Conference on*. IEEE, 2005, pp. 318b–318b.

[17] R. Nilsson, J. Offutt, and J. Mellin, "Test case generation for mutation-based testing of timeliness," *Electronic Notes in Theoretical Computer Science*, vol. 164, no. 4, pp. 97–114, 2006.

[18] P.-E. Hladik, H. Cambazard, A.-M. Déplanche, and N. Jussien, "Solving a real-time allocation problem with constraint programming," *Journal of Systems and Software*, vol. 81, no. 1, pp. 132–149, 2008.

[19] S. Di Alesio, A. Gotlieb, S. Nejati, and L. Briand, "Testing deadline misses for real-time systems using constraint optimization techniques," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 764–769.

[20] K. Apt, *Principles of constraint programming*. Cambridge University Press, 2003.

[21] P. Van Hentenryck, *The OPL optimization programming language*. MIT Press, 1999.

[22] S. Di Alesio, "The deadline misses constraints in ILOG solver v2," Tech. Rep., 2013. [Online]. Available: http://home.simula.no/~stefanod/ilog2.pdf

[23] M.-A. Peraldi-Frati and Y. Sorel, "From high-level modelling of time in MARTE to real-time scheduling analysis," *ACESMB*, p. 129, 2008.

[24] S. Anssi, S. Tucci-Piergiovanni, S. Kuntz, S. Gérard, and F. Terrier, "Enabling scheduling analysis for AUTOSAR systems," in *Object/Component/Service-Oriented Real-Time Distributed Computing, 2011 14th IEEE International Symposium on*. IEEE, 2011, pp. 152–159.

[25] K. Traore, E. Grolleau, and F. Cottet, "Simpler analysis of serial transactions using reverse transactions," in *Autonomic and Autonomous Systems, International Conference on*. IEEE, 2006, pp. 11–11.

[26] C. D. Locke, D. R. Vogel, L. Lucas, and J. B. Goodenough, "Generic avionics software specification," DTIC Document, Tech. Rep., 1990.

[27] M. Mikučionis, K. G. Larsen, J. I. Rasmussen, B. Nielsen, A. Skou, S. U. Palm, J. S. Pedersen, and P. Hougaard, "Schedulability analysis using UPPAAL: Herschel-Planck case study," in *Leveraging Applications of Formal Methods, Verification, and Validation*. Springer, 2010, pp. 175–190.

[28] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Software Engineering, 33rd International Conference on*. IEEE, 2011, pp. 1–10.