

Combining Genetic Algorithms and Constraint Programming to Support Stress Testing of Task Deadlines

STEFANO DI ALESIO, Certus Centre, Simula Research Laboratory, and SnT Centre, University of Luxembourg
LIONEL C. BRIAND, SnT Centre, University of Luxembourg
SHIVA NEJATI, SnT Centre, University of Luxembourg
ARNAUD GOTLIEB, Certus Centre, Simula Research Laboratory

Tasks in Real Time Embedded Systems (RTES) are often subject to hard deadlines, that constrain how quickly the system must react to external inputs. These inputs and their timing vary in a large domain depending on the environment state, and can never be fully predicted prior to system execution. Therefore, approaches for stress testing must be developed to uncover possible deadline misses of tasks for different input arrival times. In this paper, we describe stress test case generation as a search problem over the space of task arrival times. Specifically, we search for worst case scenarios maximizing deadline misses where each scenario characterizes a test case. In order to scale our search to large industrial-size problems, we combine two state-of-the-art search strategies, namely Genetic Algorithms (GA) and Constraint Programming (CP). Our experimental results show that, in comparison with GA and CP in isolation, GA+CP achieves nearly the same effectiveness as CP and the same efficiency and solution diversity as GA, thus combining the advantages of the two strategies. In light of these results, we conclude that a combined GA+CP approach to stress testing is more likely to scale to large and complex systems.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Experimentation, Measurement, Performance, Reliability

Additional Key Words and Phrases: Real-Time Systems, Stress Testing, Task Deadline, Search-Based Software Testing, Genetic Algorithms, Constraint Programming

ACM Reference Format:

Stefano Di Alesio, Lionel C. Briand, Shiva Nejati and Arnaud Gotlieb. 2015. Combining Genetic Algorithms and Constraint Programming to Support Stress Testing of Task Deadlines. *ACM Trans. Softw. Eng. Methodol.* V, N, Article A (January YYYY), 34 pages.

DOI : <http://dx.doi.org/10.1145/0000000.0000000>

1. 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. Specifically, widely used safety standards like IEC 61508 and IEC 26262 clearly state the importance of performance analysis for high Safety Integrity Levels [Brown 2000]. However, safety-critical systems are progressively relying on real-time embedded software

The first and fourth authors 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/P10/03 – Validation and Verification Laboratory).

Author's addresses: S. Di Alesio, Software Engineering Department, Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway; L. C. Briand, University of Luxembourg, SnT Centre, 4 rue Alphonse Weicker, L-2721 Luxembourg; S. Nejati, University of Luxembourg, SnT Centre, 4 rue Alphonse Weicker, L-2721 Luxembourg; A. Gotlieb, Software Engineering Department, Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© YYYY ACM. 1049-331X/YYYY/01-ARTA \$15.00

DOI : <http://dx.doi.org/10.1145/0000000.0000000>

that features multi-threaded application design, highly configurable operating systems, and multi-core architectures for computing platforms [Kopetz 2011]. The concurrent nature of the operating environment also entails that the order of external events triggering the system tasks is often unpredictable [Gomaa 2006]. Such complexity in system architecture, concurrency, and environment renders performance analysis and testing increasingly challenging. This aspect is reflected by the fact that most existing testing approaches target only the system functionality, though the degradation in performance can have more severe consequences than incorrect system responses [Weyuker and Vokolos 2000].

In this paper, we focus on a common [Sprunt et al. 1989] 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 approaches 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 [Gomaa 2006]. Based on estimates for task execution times, these methods are mostly used to assess the task schedulability at design time through formulas and theorems from Real-Time Scheduling Theory [Tindell and Clark 1994]. However, in practice, engineers are also interested in near deadline misses, as this analysis is based on estimated execution times. Moreover, extending these theories to multi-core processors has been shown to be a challenge [David et al. 2010]. 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, when provided with sufficient time to terminate, either report that all deadlines are met, or generate counterexamples identifying scenarios revealing deadline misses [Alur et al. 1990]. However, since model checkers are not geared toward optimization, they are not able to identify worst-case scenarios representing near deadline misses, where the nearness cannot be estimated a priori.

To complement design verification, our performance analysis approach [Nejati et al. 2012] is targeted at software testing. Indeed, note that formal verification is in general complementary to testing. In large and complex systems, it is hard to build models at the level of detail Model Checking needs to meaningfully verify a specification. On the other hand, testing a system can only prove the presence of faults, but can never prove their absence. Ideally, formal verification and testing should both be used when verifying safety-critical systems, since the two fill different roles. Specifically, the goal of our approach is to identify *worst-case scenarios* that exercise a system in a way that tasks are pushed as close as possible to their deadlines, and may possibly miss them. Consistent with the widely accepted definition [Beizer 2002], 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, search strategies are needed to 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, where each solution represents one test case. The most recent contributions in this direction that have been proposed for automated stress test case generation are based on meta-heuristics and incomplete search, namely Genetic Algorithms (GA) [Briand et al. 2006], and on complete search using Constraint Programming (CP) [Di Alesio et al. 2013; Di Alesio et al. 2014]¹.

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 stress test cases, and their power for revealing deadline misses. In our previous work [Di Alesio et al. 2013], we analyzed such tradeoffs for GA and CP through a series of experiments, and observed in some cases an opposing

¹ Note that, when confusion with the meta-heuristics and the programming paradigm can not arise, we refer to these GA-based and CP-based strategies simply as *GA* and *CP*.

trend. Specifically, GA was more *efficient*, i.e., faster in generating test cases, while CP was more *effective*, i.e., it generated test cases that were more likely to reveal deadline misses. In practice, it is also important to evaluate to what extent the test cases generated by an approach exercise different aspects of the system under test. This concept is commonly known as *test coverage*, and is an important metric to assess the quality of a test suite [Myers et al. 2011]. Investigating this aspect, we noticed in our previous experiments that GA generated a large number of test cases that were also highly *diverse*, i.e., they had a higher variety in terms of (1) time span and (2) preemptions between task executions, and (3) number of aperiodic tasks executions. On the other hand, CP generated fewer solutions which were mostly redundant. At a high level of abstraction, GA provided more coverage than CP due to the higher diversity in the test cases generated.

Therefore, the goal of this paper is to present and compare an approach (GA+CP), based on the combined use of Genetic Algorithms and Constraint Programming, holding both the efficiency and solution diversity of GA and the effectiveness of CP. The choice of combining the two search strategies has been motivated by the analysis of the results of our previous experiments. Specifically, we looked into the possibility of further improving the solutions computed by GA by performing a complete search with CP in their neighborhood. In this way, GA+CP takes advantage of the efficiency of GA, because solutions are initially computed with GA, and the subsequent CP search is likely to terminate in a short time since it focuses on the neighborhood of a solution, rather than on the entire search space. GA+CP also takes advantage of the diversity of the solutions found by GA, because CP performs a local search in subspaces defined by GA solutions. Similarly, GA+CP takes advantage of the effectiveness of CP since, once GA has found a solution, CP further improves it by either finding the best solution within the neighborhood, or proving upon termination that no better solution exists. Even though several other search strategies could have been considered, our previous experimental results [Di Alesio et al. 2013] motivated us to combine metaheuristics and constraint programming, which are based on different principles that offer distinct practical advantages. Given the existing work in the field of stress testing, GA has proven to be the best candidate for the class of meta-heuristic search strategies.

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

- (1) We propose a tool-supported, efficient and effective approach that combines GA and CP (GA+CP) to generate stress test cases that maximize the likelihood of task deadline misses.
- (2) We analyze the performance of GA+CP and compare it to the existing GA and CP alternatives through a series of experiments on five subject systems from safety-critical domains. Our experimental results show that GA+CP performs significantly better than both GA and CP in terms of quality of test cases and time required for finding them. Specifically, GA+CP achieves the same effectiveness as CP and practically the same efficiency and solution diversity as GA, thus combining the strengths of the two individual strategies. These results suggest that GA+CP is thus more likely than GA and CP in isolation to successfully scale to large industrial-scale systems.

Structure of the Paper. The rest of the paper is organized as follows. Section 2 describes the problem of investigating deadline misses among concurrent tasks, while Section 3 discusses the related work for analyzing timing properties in RTES. Section 4 presents our approach to generate stress test cases and Section 5 details how Genetic Algorithms and Constraint Programming can be combined to support stress testing of task deadlines. Finally, Section 6 details the experiment set-up and discusses the results, while Section 7 concludes the paper by summarizing the experimental results and providing some insights on potential future works.

2. PROBLEM DESCRIPTION

Real-Time Embedded Systems (RTES) 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 [Gomaa 2006]. Moreover, RTES are often safety critical, and thus bound to meet strict performance requirements [Kopetz 2011]. In addition, their tasks must satisfy execution constraints with respect to dependencies such as 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 1.1 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 . 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. In the example, time is discretized in ten *time quanta*, and the durations of tasks are expressed as multiples of a time quantum. Specifically, j_0 , j_1 , and j_2 have a duration of respectively 2, 3, and 2 time quanta. The three tasks have to complete before their respective deadlines dl_0 , dl_1 , and dl_2 , which are static and specified with respect to the task arrival times. In the example system, j_0 , j_1 , and j_2 have to respectively finish their execution 6, 4, and 3 time quanta after their arrival. Note that, in RTES, task deadlines are usually fixed prior to execution and specified in the system requirements. For periodic tasks, the deadline is usually equal to the period, to avoid consecutive executions of the same task to stack without terminating. For aperiodic tasks, the deadline usually depends on considerations about the system domain. This is because aperiodic tasks correspond to external triggers that often represent alarm signals. To be deemed safe, the system has to react to these triggers within a given amount of time, which is specified by task deadlines. Also note that, in this example, the task deadlines are inversely proportional to their priority. This is common in RTES, since higher priority tasks usually implement critical functionalities that have to be completed within short time [Audsley et al. 1991].

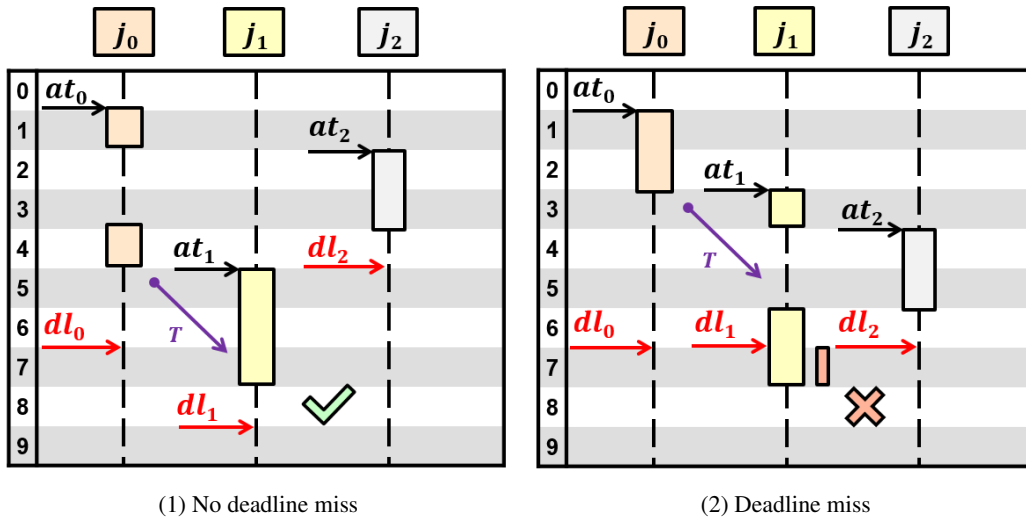


Fig. 1: Impact of changes in the arrival times of tasks with respect to deadline miss properties

Figure 1 reports 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 1.1, 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 1.2 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 prior 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, which may evolve over time, and cannot always be fully predictable prior to the execution of the system. In order to evaluate deadline miss constraints, we need a strategy to search for 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 refer to these scenarios as *worst-case scenarios* with respect to task deadlines, and 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. Note that each stress test case represents a single worst-case scenario with respect to task deadlines, and, to search for these worst-case scenarios, we consider the duration of each task as its Worst-Case Execution Time (WCET). This is common when analyzing task real-time properties in the worst case [Gomaa 2006]. To efficiently and effectively drive the search towards such worst-case scenarios, we have implemented a search strategy (Section 5.2), based on a combination of two existing approaches based on Genetic Algorithms [Briand et al. 2006] and Constraint Programming [Di Alesio et al. 2013].

3. RELATED WORK

Testing multi-threaded concurrent software has largely focused on functional properties rather than on performance requirements [Weyuker and Vokolos 2000]. In RTES, performance properties have mostly been studied through verification approaches, such as Schedulability Theory [Tindell and Clark 1994] and Model Checking [Alur et al. 1990], rather than testing. Even though our approach can also be used for design-time verification, the focus of this paper is stress testing, intended as the generation of scenarios that are likely to break task deadlines. For this reason, Schedulability Theory and Model Checking are not comparable to our work as they are not meant to generate scenarios, but rather to assess whether a system model meets its specification. Specifically, theorems from Schedulability Theory are limited to providing sufficient or necessary conditions for a set of tasks to be schedulable [Baker 2006]. Model Checking approaches instead analyze time-related properties, such as task deadlines and resource usage, by proving reachability properties in state machines [David et al. 2010]. When these properties do not hold, such approaches are able to provide counter-examples similar to the scenarios that are the focus of our study. However, Model Checking faces limitations when it comes to generating worst-case scenarios with respect to time-related properties such as task deadlines. First, Model Checking requires complex formal modeling of the system, which often leads to the well-known state explosion problem that has not been solved in the general case [Clarke et al. 2012]. Second, upon termination, Model Checking approaches generate counterexamples that represent violations of real-time properties. In practice, engineers are also interested in deadline near-misses, i.e., scenarios where tasks are predicted to be close to missing a deadline by a certain amount of time (*nearness*). Such scenarios provide engineers with valuable insights into the robustness of their systems. Model checkers are not geared towards optimization, and hence require additional effort to identify worst-case scenarios representing these deadline near-misses. In particular, to generate counterexamples with near-misses, the nearness would have to be quantified a priori. Some recent advancements (e.g., the Ptolemy project²) attempt to adapt model checkers to provide a measure of fitness, as opposed to a proof of correctness [Henzinger 2013]. However, the overall scalability of model checkers to large realistic real-time systems is still unknown.

Recall from Section 2 that, in this paper, we consider the problem of identifying worst-case scenarios with respect to task deadlines. Note that this is a different problem from identifying task Worst-Case Execution Times (WCET). Indeed, a task WCET is defined as the maximum time that tasks are executing, and hence keep the CPU busy [Wilhelm et al. 2008]. This definition does not take into account the time during which a task is preempted by other tasks or during which it is blocked waiting for computational resources, because in these cases the task is not using the CPU.

² <http://ptolemy.eecs.berkeley.edu/>

However, estimating the task WCET is a prerequisite for the successful application of our test case generation approach, because in our deadline misses analysis (Section 5) we assume the duration of each task to be equal to its WCET.

In industrial contexts, the problem of ensuring that system tasks meet their deadlines is mostly studied by Performance Engineering techniques, which extensively rely on profiling and benchmarking tools to dynamically analyze performance properties [Jain 2008]. Such tools, however, are limited to producing a small number of system executions and require manual inspection of those executions. In general, these tools provide only a rough assessment of system performance, and cannot replace systematic stress and performance testing. Recently, there has been an attempt at automating stress testing through the use of PID controllers [Bayan and Cangussu 2008]. These controllers implement feedback control loops that dynamically adjust the system inputs in order to maximize resources consumption. However, in such an approach, the controllers closely depend on the target system implementation to analyze inputs and outputs.

Over the years, there has been a growing interest in using model-based approaches for performance testing, especially in the domains of distributed systems [Del Grosso et al. 2005; Shams et al. 2006; Barna et al. 2011]. In our prior work [Nejati et al. 2012] 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. Then, we focused on the problem of generating stress test cases to break task deadlines [Di Alesio et al. 2012], and improved our constraint model by devising and implementing heuristics to significantly speed up the search process [Di Alesio et al. 2013]. We evaluated our constraint program by systematically comparing it with a state-of-the-art Genetic Algorithm (GA) search strategy [Briand et al. 2006], and then refined it by improving its data structures [Di Alesio et al. 2014]. This paper builds upon our earlier work, improving it in the following respects.

- (1) We have devised and implemented a combined approach, namely GA+CP, to generate stress test cases that are likely to reveal task deadline misses in such a way as to benefit from the strengths of each individual search strategy. To the best of our knowledge, there has been no prior attempt at combining GA and CP to support stress test case generation.
- (2) We evaluated our work by systematically comparing it with state-of-the-art GA [Briand et al. 2006] and CP [Di Alesio et al. 2013] search strategies.
- (3) As a result, our approach has a remarkably better efficiency than CP, in terms of the time needed to generate test cases, better effectiveness than GA, in terms of revealing power for deadline misses, and better diversity than CP, in terms of redundancy in test cases. In effect, these results clearly show that GA+CP combine the strengths of GA and CP.

Search-based approaches have extensively been used to test non-functional system properties [Afzal et al. 2009]. Specifically, GA have successfully been used to support performance testing in the domain of distributed systems with respect to network traffic [Garousi et al. 2008], QoS constraints [Shams et al. 2006], and computational resources consumption [Berndt and Watkins 2005]. However, these approaches do not tackle hard real-time constraints such as deadline misses, as these properties are mostly important in Real-Time Systems. In this domain, GA have also been used to generate test cases for testing timeliness properties, showing that they are able to run in large systems where Model Checking approaches were not able to run [Nilsson et al. 2006]. However, the main contribution of Nilsson et al. is a mutation-based testing criterion to assess test adequacy, which is beyond the scope of this paper. As for stress testing task deadlines, the state-of-the-art is represented by the work of Briand et al. [Briand et al. 2006], which we used in this paper as the GA part of our combined approach.

CP has been applied for a long time to solve schedulability analysis problems [Baptiste et al. 2001], especially in the domain of job-shop scheduling [Le Pape and Baptiste 1997]. Among those, several approaches target task real-time constraints such as task deadlines [Hladik et al. 2008], or timeliness [Malapert et al. 2012]. Preemptive scheduling problems have also been solved both with pure CP [Cambazard et al. 2004], and with hybrid approaches combining GA with complete search [Yun and Gen 2002]. Furthermore, recent implementations [Laborie 2009] have successfully

used the IBM ILOG CPLEX CP OPTIMIZER and OPL for scheduling problems, albeit not addressing task preemption. However, the goal of these schedulability analysis approaches is to assess whether or not the system tasks are schedulable, i.e., to find scenarios where tasks do not miss their deadlines. The goal of this work is the opposite, as we are interested in generating scenarios that break task deadlines. In the context of stress testing, CP has been used to generate stress test cases for multimedia systems [Zhang and Cheung 2002]. The goal of that work is to investigate resource saturation at runtime under heavy loads, as opposed to task deadlines.

There have been various contributions in the area of hybrid search strategies to solve hard combinatorial problems [Raidl 2006], especially in the direction of combining CP with probabilistic meta-heuristics [Focacci et al. 2003]. For example, GA has been used in combination with CP to drive a complete search towards optimal solutions [Homaifar et al. 1994]. There has also been interest in studying how to combine CP and Local Search (LS) [Hentenryck and Michel 2009]. Most of these approaches compute a set of initial solutions at random, and then optimize them by exploring their neighborhood [Mladenović and Hansen 1997]. For instance, Large Neighborhood Search [Shaw 1998] systematically explores subpart of the search space by relaxing current sub-optimal solutions and using constraint propagation to find better solutions. Despite this large number of hybrid search strategies, we are unaware of applications that are targeted at testing timing properties by generating stress test cases. However, the search strategy presented in our work shares several commonalities with existing work. Specifically, we use CP to completely explore a neighborhood of a solution computed externally, similar to the strategy proposed by Pesant et al. to solve the Traveling Salesman Problem [Pesant and Gendreau 1996]. In particular, we use CP to improve a set of solutions initially computed by GA, which is a principle similar to that of memetic algorithms, where LS is used in the neighborhood of solutions found by GA [Harman and McMinn 2010]. Note that memetic algorithms have successfully been used for the purpose of generating test suites that optimize branch coverage [Fraser et al. 2014]. As opposed to memetic algorithms, we use CP in place of LS to optimize the solutions found by GA. Guimarans et al. [Guimarans et al. 2011] used the Clark and Wright Savings Heuristic to generate an initial set of solutions to further optimize with CP, in a similar way as we use the GA strategy proposed by Briand et al. [Briand et al. 2006] to initially generate solutions. We finally point out how, in contrast to strategies where GA is used as a mean to explore the CP search tree [Iwamura and Liu 1996], GA and CP are independent in our approach, as the latter is used only once the solutions have been found by the former. Note that approaches where metaheuristic search and CP have been used independently have already proven to be successful in the area of software testing. For example, Lakhotia et al. proposed a test data generation approach aimed at optimizing branch coverage [Lakhotia et al. 2010]. In that work, metaheuristic search is used to optimize numerical inputs, while constraint solving is used to find solutions for pointer inputs.

4. APPROACH OVERVIEW

At a high level, the approach presented here builds upon our earlier work [Nejati et al. 2012] for deriving test cases exercising the CPU usage requirements of a real-time system 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, as depicted by 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 5. For their mapping to the UML/MARTE profile we refer the reader to our earlier work [Nejati et al. 2012] as this is not the focus of this paper.

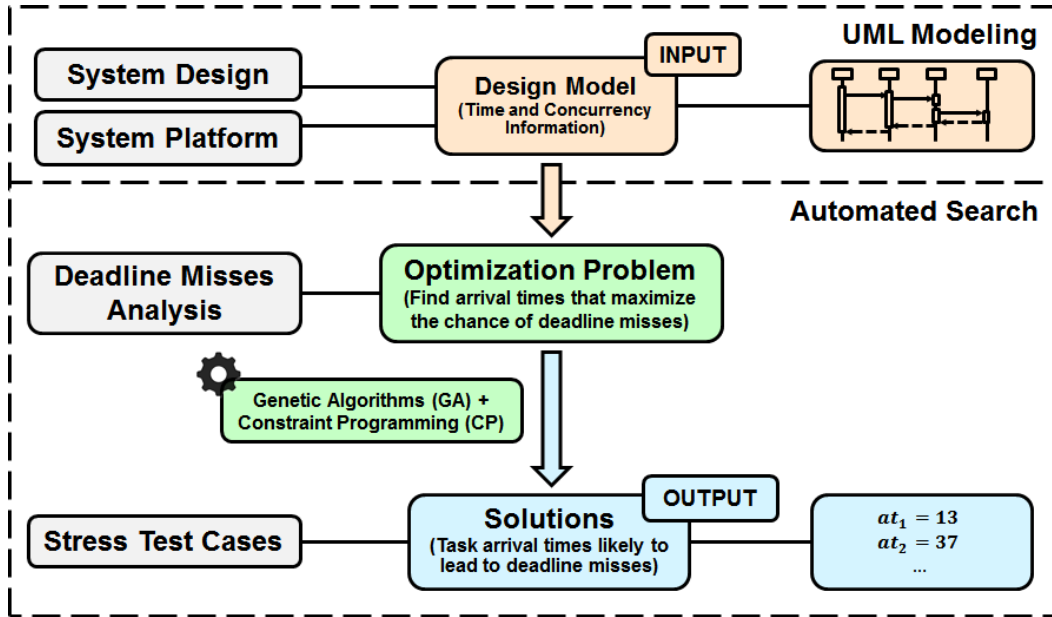


Fig. 2: Our approach for schedulability risk analysis in RTES

The analysis of deadline misses is cast as an optimization problem over the abstractions represented in the design model, that defines the search space over which to optimize. 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 a combination of Genetic Algorithms and Constraint Programming. 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.

5. DEADLINE MISS ANALYSIS

In order to describe our deadline miss analysis, we first define the necessary abstractions in Section 5.1 using the notation from our previous work [Di Alesio et al. 2013]. These abstractions enable the definition of our combined GA+CP search strategy, presented in Section 5.2. Finally, we show a working example of the proposed search strategy in Section 5.3.

We point out that 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.
- (2) The interval of time in which the scheduler switches context between tasks is negligible compared to a time quantum.

These two assumptions are realistic 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 [Singh 2009].

5.1. Timing and Concurrency Abstractions

Our deadline miss analysis is based on the definition of the following timing and concurrency abstractions.

- **Observation Interval T .** 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,

implying that time is discretized in our analysis. We refer to each time value $t \in T$ as a *time quantum*. In Figures 1.1 and 1.2, $T = [0, 9]$: this means that $tq = 10$, and therefore T includes 10 time quanta.

- **Number c of platform cores.** Let c be an integer number representing the number of cores in the execution platform. By definition, c represents the maximum number of tasks that are allowed to be executed in parallel. In Figures 1.1 and 1.2 we assumed $c = 1$, as tasks are not 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, and a set of *dynamic* properties, where each property is represented by an integer value. Let J_p and J_a be the set of periodic and aperiodic tasks of the system, respectively. J_p and J_a define a partition over J , i.e.: $J_p \cap J_a = \emptyset$ and $J_p \cup J_a = J$. In our work, 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 analysis. In Figure 1.1, $J = J_a = \{j_0, j_1, j_2\}$, and $J_p = \emptyset$.
- **Static Properties of Tasks.** The static properties express temporal requirements and constraints that regulate task execution, and are defined in Real-Time Scheduling Theory [Sprunt et al. 1989]. These properties depend on the system requirements and design, and hence are known prior to the analysis.
 - *priority*(j), the priority of task j . For example, in Figure 1.1, *priority*(j_0) = 0, *priority*(j_1) = 1, and *priority*(j_2) = 2.
 - *period*(j), the period of task j . Only defined if j is periodic.
 - *offset*(j), 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. Only defined if j is periodic.
 - *min_ia_time*(j) and *max_ia_time*(j), the minimum and maximum inter-arrival times, respectively, i.e., the minimum and maximum time separating two consecutive arrival times of task j . Only defined if j is aperiodic since for all periodic tasks j , *min_ia_time*(j) = *max_ia_time*(j) = *period*(j) holds. For example, in Figures 1.1 and 1.2, $\forall j \in J \cdot \text{min_ia_time}(j) = 10$.
 - *duration*(j), the estimated Worst Case Execution Time (WCET) of task j . For simplicity, we define the integer interval P_j of task execution slots as $P_j \stackrel{\text{def}}{=} [0, \text{duration}(j) - 1]$. In Figure 1.1, *duration*(j_0) = 2 and $P_{j_0} = [0, 1]$.
 - *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 $\forall j \in J_p \cdot \text{deadline}(j) \leq \text{period}(j)$. In Figure 1.1, *deadline*(j_0) = 6.
- **Dynamic Properties of Tasks.** The dynamic properties express variables that depend on the runtime behavior of the system, and hence are not known prior to the analysis. Indeed, the values for these variables are calculated during the search (Section 5.2) for worst-case scenarios. We partition dynamic properties into *independent* and *dependent* properties. The independent properties characterize stress test cases in terms of the events that trigger the task executions. On the other hand, the dependent properties characterize the expected reaction of the system to the events modeled by the independent properties, in terms of the way tasks execute. The dependent properties are formally defined as expressions of static and independent properties, and are meant to simplify our notation. Note that the search process manipulates only the values of independent properties which, once fixed, determine the value of dependent properties. Therefore, a *solution* found by our search strategy is an assignment of values to independent properties.
 - **Independent Properties.** We define two independent properties, concerning the number of times tasks are executed, and task arrival times, respectively.
 - *task_executions*(j), the number of times task j is executed within T . For simplicity, we define the integer interval K_j of task executions for task j as $K_j \stackrel{\text{def}}{=} [0, \text{task_executions}(j) - 1]$. Furthermore, we refer to the k^{th} execution of task j as the couple (j, k) . We assume the offset and period bound the number of executions

of periodic tasks: $\forall j \in J_p \cdot task_executions(j) = \lfloor \frac{T - offset(j)}{period(j)} \rfloor$. Similarly, we assume the minimum and maximum inter-arrival times bound the number of executions of an aperiodic task, implying that $\forall j \in J_a \cdot \lfloor \frac{T}{max_ia_time(j)} \rfloor \leq task_executions(j) \leq \lfloor \frac{T}{min_ia_time(j)} \rfloor$. In Figures 1.1 and 1.2 each task has only one execution, therefore $task_executions(j_0) = task_executions(j_1) = task_executions(j_2) = 1$ and $K_{j_0} = K_{j_1} = K_{j_2} = \{0\}$.

- $arrival_time(j, k)$, the time when an event notifies the scheduler that task j should be executed for the k^{th} time. We say that j arrives for the k^{th} time at time t iff $arrival_time(j, k) = t$. 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 $\forall j \in J_p, k \in K_j \cdot arrival_time(j, k) = offset(j) + k \cdot period(j)$. In the case where $\forall j \in J \cdot offset(j) = 0$, our assumption is the same one made by the Generalized Completion Time Theorem (GCTT) [Gomaa 2006] to ensure that the analysis considers the case where all periodic tasks simultaneously arrive for the first time at the beginning of T . Furthermore, we assume that $\forall j \in J_a, k \in K_j \setminus \{0\} \cdot arrival_time(j, k - 1) + min_ia_time(j) \leq arrival_time(j, k) \leq arrival_time(j, k - 1) + max_ia_time(j)$. This assumption is consistent with the definition of minimum and maximum inter-arrival times as the intervals of time separating two consecutive arrival times of j . In the case where $k = 0$, we assume that $\forall j \in J_a \cdot 0 \leq arrival_time_0(j) \leq max_ia_time(j)$. In this way, we ensure that each task arrival time corresponds to a task execution, and vice-versa. In Figure 1.1, $arrival_time(j_0, 0) = 1$.
- **Dependent Properties.** We define six dependent properties, four related to the execution pattern of the system tasks, and two related to their deadlines. The first four properties concern the time quanta when tasks are executing, the time quanta when tasks are preempted by higher priority tasks, and the task start and end times. These properties enable the analysis of the way system tasks execute when responding to external events, and are necessary for the formal definition of the concept of diversity (Section 6.3.2) when comparing solutions found by search strategies. The last two properties represent the task absolute deadlines, and the number of time quanta by which tasks miss their deadline, which enable the definition of the function that drives the search towards optimal solution.
 - $active(j, k, p)$, the p^{th} time quantum in T in which task j is running for the k^{th} execution. We assume that the executing time slots $p \in P_j$ define an order over the executing time quanta for a task j : $\forall j \in J, k \in K_j, p \in P_j \setminus \{0\} \cdot active(j, k, p - 1) < active(j, k, p)$. We further assume that each task starts being executed after its arrival time: $\forall j \in J, k \in K_j, p \in P_j \cdot arrival_time(j, k) \leq active(j, k, p)$. For simplicity, we define the integer vector $A_{j,k}$ of time quanta where j is executing for the k^{th} time as $A_{j,k} = [active(j, k, p) \mid p \in P_j]$, the vector A_j of time quanta where j is executing as $A_j = [A_{j,k} \mid k \in K_j]$, and the vector A of time quanta where tasks are executing as $A = [A_j \mid j \in J]$. Note that A_j and A are defined as vectors of vectors. We also refer to A as the *schedule* produced by the arrival times of the tasks in J . In Figure 1.1, $active(j_0, 0, 0) = 1$, $active(j_0, 0, 1) = 4$, $A_{j_0,0} = [1, 4]$, $A_{j_0} = [[1, 4]]$, and $A = [[1, 4], [5, 6, 7], [2, 3]]$.
 - $preempted(j, k, p)$, the amount of time quanta for which the k^{th} execution of task j is preempted for the p^{th} time: $preempted(j, k, p) \stackrel{\text{def}}{=} active(j, k, p) - active(j, k, p - 1) - 1$. Only defined for $p > 0$. In Figure 1.1, $preempted(j_0, 0, 1) = 2$.
 - $start(j, k)$ and $end(j, k)$, the first and the one after the last time quantum in which j is executing for the k^{th} time, respectively: $start(j, k) \stackrel{\text{def}}{=} active(j, k, 0)$ and $end(j, k) \stackrel{\text{def}}{=} active(j, k, duration(j) - 1) + 1$. In Figure 1.1, $start(j_1, 0) = 5$ and $end(j_1, 0) = 8$.
 - $task_deadline(j, k)$, the absolute deadline of the k^{th} execution of j , i.e., the time, with respect to the beginning of T , before which j should terminate to meet its

deadline: $task_deadline(j, k) \stackrel{\text{def}}{=} arrival_time(j, k) + deadline(j) - 1$. In Figure 1.1, $task_deadline(j_0, 1) = 6$.

- $deadline_miss(j, k)$, the amount of time by which j missed its deadline during its k^{th} execution, i.e., $deadline_miss(j, k) \stackrel{\text{def}}{=} end(j, k) - task_deadline(j, k) - 1$. Negative if $end(j, k) - 1 < task_deadline(j, k)$, non-negative otherwise. In Figure 1.1, $deadline_miss(j_0, 1) = -2$.

We use an alternative notation for the dynamic properties when making their context explicit is required. In such notation, the meta-variables j , k and p are reported as subscripts, and the parentheses contain the specific context that the dynamic properties refer to, such as the system under analysis, the search strategy used, or the solution the properties belong to. For instance, we write $start_{j,k}(x)$ to mean the value of $start(j, k)$ in the solution x .

- **Relationships between Tasks.** Tasks can depend on other tasks to implement functionalities such as inter-process communication, and sequential actions. These task dependencies are captured through relationships which are defined at design time, and hence, similar to static properties, are known prior to the analysis. Let the following be three binary relations defined over $J \times J$.
 - $triggers(j_1, j_2)$ holds if the event triggering the task j_2 occurs when the task j_1 finishes its execution, i.e. $\forall k \in K_{j_1} \cdot arrival_time(j_2, k) = end(j_1, k)$. In Figure 1.1, $triggers(j_0, j_1)$ holds. The relation $triggers$ is defined as irreflexive and antisymmetric. Tasks triggering other tasks upon termination are common in RTES, especially in cases where functionalities are implemented by a sequential chain of procedures, each starting when the previous has finished [Liu et al. 2000].
 - $dependent(j_1, j_2)$ 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. In complex RTES applications, tasks often use locks on critical sections to implement communication mechanisms or to guarantee atomicity of read/write operations [Liu et al. 2000].
 - $impacts(j_1, j_2)$ holds if the arrival time of an execution of j_1 can have a *direct* impact over an execution of j_2 . In practice, this can happen in two cases. The first case occurs if j_1 has higher priority than j_2 . In this case, j_2 can be preempted at any time by j_1 if there are not enough cores available. The second case occurs when j_1 and j_2 share the same computational resource. In this case, if j_1 arrives before j_2 has acquired the lock on the resource, the latter will have to wait until such lock is released. Therefore, we define the relation $impacts$ in the following way: $impacts(j_1, j_2) \stackrel{\text{def}}{=} priority(j_1) \geq priority(j_2) \vee dependent(j_1, j_2)$. Note that $impacts$ is reflexive, because trivially the arrival time of a task has an impact over the execution of the task itself. However, j_1 having higher priority or depending on j_2 is not the only case in which j_1 can have an impact over an execution of j_2 . For instance, consider the case where j_1 has higher priority than another task j_3 , and j_3 depends on j_2 . In this case, j_1 could preempt j_3 after j_3 has acquired the lock on the resource shared with j_2 , and then j_2 would have to wait for both the completion of j_1 and j_3 before being able to execute. In this case, j_1 can also have an impact over an execution of j_2 , albeit this impact is *indirect* since it involves j_3 . For this reason, we define $impacts^+$ as the transitive closure of $impacts$ to cover all the cases where j_1 can have a direct or indirect impact over the execution of j_2 .
- **Impacting Set I of a Task.** Let $I : J \rightarrow \mathcal{P}(J)$ be the function that represents the set of tasks that can have an impact over the execution of a target task: $I_j \stackrel{\text{def}}{=} \{j' \in J \mid impacts^+(j', j)\}$. We refer to I_j as the *impacting set* of j .
- **Performance Requirement.** As explained in Section 4, 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 an *objective function* in the context of Constraint Programming, and as a *fitness function* in the context of Genetic Algorithms. In this paper, we adopted the function F from our previous work [Di Alesio et al. 2013]:

$$F = \sum_{j \in J, k \in K_j} 2^{\text{deadline_miss}(j,k)} \quad (1)$$

F is formulated in a way that allows it to be effective for stress testing purposes by meeting three important characteristics. Being a sum of exponentials, F guarantees that (1) no deadline miss is overshadowed by executions of the same task that finish long before their deadline, (2) the more deadline misses, the higher the function value, and, (3) the larger the deadline misses, the higher the function value [Di Alesio et al. 2013].

5.2. Combining Genetic Algorithms and Constraint Programming to Support Stress Testing

In this paper, we present a hybrid approach that combines Genetic Algorithms (GA) and Constraint Programming (CP) for the purpose of generating sequences of arrival times likely to break task deadlines. The approach is based on the definition of an optimization problem, where each solution represents a worst-case scenario with respect to task deadlines, and therefore characterizes a stress test case.

GA have been used in the past to support stress testing of task deadlines. Among others, Briand et al. proposed a GA strategy [Briand et al. 2006] to generate sequences of arrival times likely to lead to deadline misses. In that work, arrival times of aperiodic tasks are modeled as chromosomes. The initial population of these chromosomes is initialized with random values, and their fitness is evaluated by computing the task schedule that is produced from the arrival times encoded in the chromosomes. Indeed, such a schedule contains information about the end times of tasks, that define the fitness function in a fashion similar to that of F . At each iteration of GA, a pair of chromosomes is crossed over and then mutated using specific operators that ensure compliance with the inter-arrival times of aperiodic tasks.

CP has also been used to automate the generation of stress test cases to break task deadlines. In our previous work [Di Alesio et al. 2014], we propose a constraint optimization model for the purpose of generating sequences of arrival times likely to lead to deadline misses. We model the static and dynamic properties of the system as integer constants and variables respectively, and we model the RTOS scheduler as a set of constraints among such variables. The constraint model is then solved using IBM ILOG CPLEX CP OPTIMIZER³, one of the leading CP solvers in the market. Experimental results [Di Alesio et al. 2013] have shown in some cases an opposing trend: while GA was more *efficient*, i.e., faster in generating test cases, CP was more *effective*, i.e., it generated test cases that were more likely to identify deadline misses. Furthermore, GA was able to find a larger and more diverse set of test cases than CP, exercising the system in a more *diverse* way with respect to task executions. Specifically, the test cases generated by GA had a higher variety in terms of (1) time span and (2) preemptions between task executions, and (3) number of aperiodic task executions. These three criteria are described in Section 6.3 and define the concept of test case *diversity*. On the other hand, CP generated a smaller number of test cases, most of which were redundant with respect to the three criteria, i.e., that executed the systems tasks during similar time intervals, with the same preemptions and number of executions.

Therefore, we looked into a way to achieve both the efficiency of GA and the effectiveness of CP. The key idea behind our work is to improve the solutions computed by GA by performing a complete search with CP in their neighborhood, hence defining a 2-stage GA+CP strategy. In particular, given that GA is more efficient (generates results faster) and CP is more effective (generates better results), the most natural choice to retain the advantages of both is to keep the solutions generated by the

³ <http://www.ibm.com/software/commerce/optimization/cplex-cp-optimizer/>

most efficient strategy, and try to improve them with the solutions generated by the most effective strategy. In this way, we expected the combined GA+CP strategy to take advantage of the efficiency of GA, because solutions are initially computed with GA and the subsequent CP search is likely to terminate in a short time since it focuses on the neighborhood of a solution, rather than in the whole search space. Similarly, we expected GA+CP to take advantage of the solutions diversity of GA, because these solutions define in turn subspaces where CP searches for better solutions. Finally, we also expected GA+CP to take advantage of the effectiveness of CP since, after GA computes a solution, CP either finds the best solution within the neighborhood, or proves upon termination that no better solution exists in such neighborhood. Figure 3 illustrates how GA+CP searches for solutions through an abstract example.

- (1) **GA Step.** The initial population of GA consists of the three solutions x_1 , y_1 , and z_1 . In the first generation, GA finds the solutions x_2 , y_2 , and z_2 that are respectively generated from x_1 , y_1 and z_1 . After the five generations, GA converges on the solutions x_6 , y_6 and z_6 .
- (2) **CP Step.** CP then searches for better solutions in the neighborhoods of x_6 , y_6 and z_6 . This step is performed by launching three separate instances of CP, each having x_6 , y_6 and z_6 as a starting point. The first two complete searches find the solutions x^* and y^* . The last proves upon termination that z_6 is the best solution in its neighborhood, hence $z_6 = z^*$. Therefore, x^* , y^* , and z^* are the final solutions found by GA+CP, and are used to characterize stress test cases.

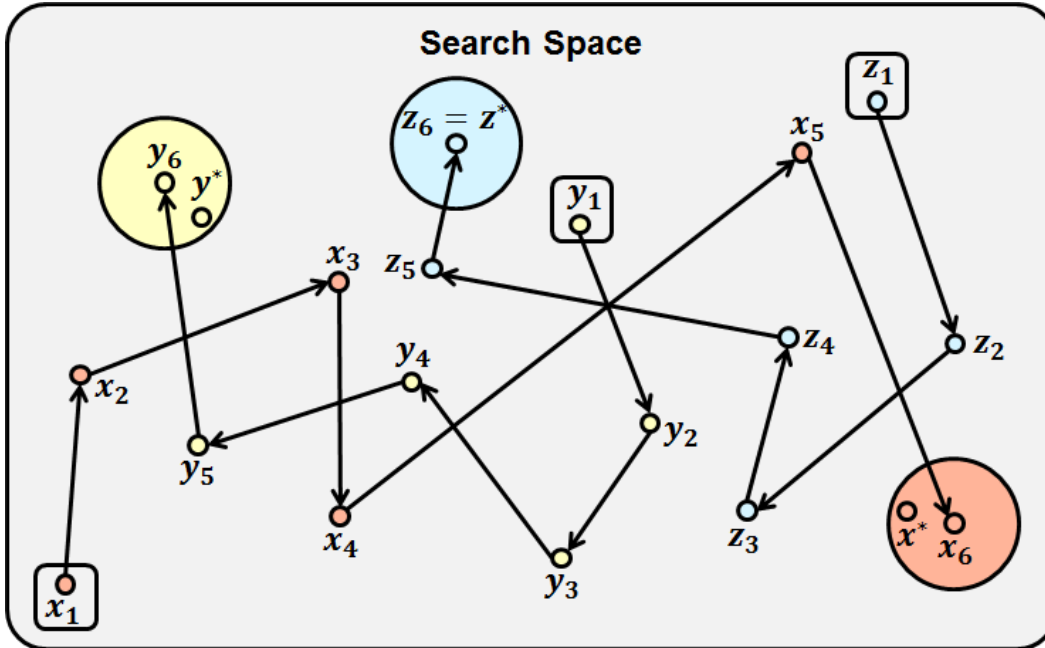


Fig. 3: Overview of GA+CP: the solutions x_1 , y_1 and z_1 in the initial population of GA evolve into x_6 , y_6 , and z_6 , then CP searches in their neighborhood for the optimal solutions x^* , y^* and z^* .

We begin the formal description of our approach with the following definitions.

- **Solution x computed by GA.** Recall from Section 4 that a solution is a sequence of arrival times which the search identifies as likely to break task deadlines, and hence characterizes a stress test case. Let $x = [[x_{j,k} \mid k \in K_j] \mid j \in J_a]$ be a solution computed by GA. Note that, by definition, x is an assignment of arrival times for aperiodic tasks, where $x_{j,k}$ is the value for the k^{th} arrival time of task j : $\forall j \in J_a, k \in K_j \cdot arrival_time_{j,k}(x) = x_{j,k}$.

- **Set J^* of tasks that miss a deadline or are the closest to missing it among all tasks.** Let $J^*(x)$ be the set of tasks that miss at least a deadline in one execution, or are closer than others to doing so in the schedule generated by the arrival times in x :

$$J^*(x) \stackrel{\text{def}}{=} \left\{ j \in J \mid \exists k^* \in K_{j^*}(x) \cdot (deadline_miss_{j^*,k^*}(x) \geq 0 \vee \forall j \in J, k \in K_j \cdot deadline_miss_{j^*,k^*}(x) \geq deadline_miss_{j,k}(x)) \right\}$$

- **Union set I^* of impacting sets of tasks missing or closest to miss their deadlines.** Let $I^*(x)$ be the union of the impacting sets of tasks in $J^*(x)$:

$$I^*(x) \stackrel{\text{def}}{=} \bigcup_{j^* \in J^*(x)} I_{j^*}(x)$$

By definition, $I^*(x)$ contains all the tasks that can have an impact over a task that misses a deadline or is closest to a deadline miss.

- **Neighborhood ε of an arrival time and neighborhood size D .** Let $\varepsilon(x_{j,k})$ be the interval centered in the arrival time $x_{j,k}$ computed by GA, and let D be its radius: $\varepsilon(x_{j,k}) = [x_{j,k} - D, x_{j,k} + D]$. ε defines the part of the search space around $x_{j,k}$ in which to find arrival times that are likely to break task deadlines. D is a parameter of the search.
- **Constraint Model \mathcal{M} implementing a Complete Search Strategy.** Let \mathcal{M} be the constraint model defined in our previous work [Di Alesio et al. 2014] for the purpose of identifying arrival times for tasks that are likely to lead to deadline miss scenarios. \mathcal{M} models the static and dynamic properties of the software system as constants and variables respectively, and the scheduler of the operating system as a set of constraints among such variables. Note that \mathcal{M} implements a complete search strategy over the space of arrival times. This means that \mathcal{M} searches for arrival times of all aperiodic tasks within the whole interval T .

Our combined GA+CP strategy consists of the following two steps:

- (1) **GA Step.** Run GA for a given amount of time to obtain a set X of solutions. For this purpose, we use the implementation of GA introduced by Briand et al., with the same initial population size, replacement strategy, and probability values used for crossover and mutation [Briand et al. 2006].
- (2) **CP Step.** For each solution $x \in X$, solve a constraint model $\mathcal{M}'(x)$ that searches for arrival times only within a fixed-size neighborhood of x . $\mathcal{M}'(x)$ is derived from \mathcal{M} by:
 - *Fixing the arrival time of tasks not in I^* .* This is done by adding to \mathcal{M} the following constraint: $\forall j \in J \setminus I^*(x), k \in K_j(x) \cdot arrival_time(j,k) = x_{j,k}$. In practice, this means that the arrival time of all task executions that do not have any impact on tasks being close to missing a deadline will be fixed in \mathcal{M}' .
 - *Bounding the arrival times of tasks in I^* .* This is done by adding to \mathcal{M} the following constraints: $\forall j \in I^*(x), k \in K_j(x) \cdot x_{j,k} - D \leq arrival_time(j,k) \leq x_{j,k} + D$. In practice, this means that the arrival time of the task executions that can have an impact on tasks identified by GA as close to miss a deadline will be declared in \mathcal{M}' as a variable with domain $\varepsilon(x_{j,k})$.

Note that, by definition, \mathcal{M}' implements a local search strategy over the space of arrival times. This means that \mathcal{M}' searches only for arrival times of aperiodic tasks that can have an impact on tasks GA identifies as close to miss a deadline, and bounds the search within a neighborhood of size D from the solution computed by GA. Specifically, for given j and k , the inequality constraints on the variable *arrival_time* define in the solutions space a hypercube of side $2D$ centered in $x_{j,k}$.

The search in GA+CP can be configured through two sets of parameters, one related to GA and the other to CP. GA relies on parameters specific to evolutionary algorithms, such as the initial population size, the crossover and mutation probabilities, and the population replacement rate. For

those, we used values that have been derived from the GA literature and specifically tuned for deadline misses analysis [Briand et al. 2006]. On the other hand, CP is a deterministic search strategy, and therefore does not require us to set such parameters. However, our combined search strategy depends on the neighborhood size D that CP searches for arrival times of aperiodic tasks. Our preliminary experimentation showed that a value of D around 1% of T yields a good compromise between efficiency and effectiveness. Specifically, lower values for D define a smaller neighborhood where GA+CP is less likely to improve the solutions found by GA, while higher values for D define a larger neighborhood where GA+CP is likely to spend a significant amount of time without finding better solutions.

5.3. GA+CP in Practice: a Working Example

In this section, we introduce an example to show how GA+CP works in practice. Consider the system composed of five aperiodic tasks detailed in Table 1. Note that *period* and *offset* are not defined, since each task is aperiodic. There are also no *triggers* relationships between tasks.

	<i>priority</i>	<i>duration</i>	<i>min_ia</i>	<i>max_ia</i>	<i>deadline</i>	<i>dependency</i>
j_0	0	2	10	10	8	
j_1	1	2	10	10	6	j_4
j_2	2	2	10	10	5	
j_3	3	2	10	10	4	
j_4	4	2	10	10	3	j_1
$T = [0, 9]$				$c = 1$		

Table 1: Example system with four tasks and one dependency

Suppose GA finds the solution $x = [[0][2][3][6][3]]$, where each task is executed once, and j_0 arrives at time 0, j_1 at time 2, j_3 at time 3, j_4 at time 6 and j_5 at time 3. The schedule corresponding to this solution is shown in Figure 4.1.

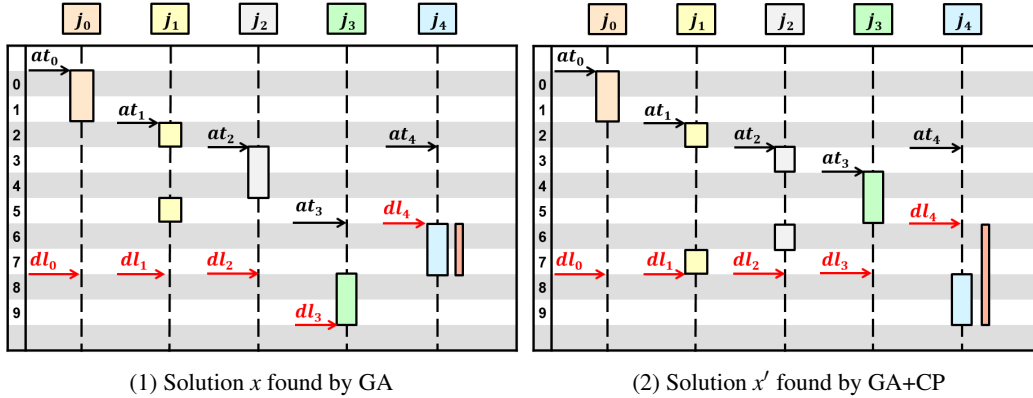


Fig. 4: GA+CP neighborhood search example

In x , j_4 misses its deadline by 2 time quanta, being the task that has the biggest deadline miss. Therefore, $J^*(x) = \{j_4\}$. By looking at the task priorities and dependencies, j_1 depends on j_4 , and both j_2 and j_3 have higher priority than j_1 , so all three can potentially have an impact over the execution of j_4 for the reasons detailed in Section 5.1. This means that $I_{j_4}(x) = \{j_1, j_2, j_3, j_4\}$, and consequently $I^*(x) = \{j_1, j_2, j_3, j_4\}$.

GA+CP searches the space around x up to a distance D from the tasks in $I^*(x)$, i.e., the tasks that can potentially have an impact over the execution of j_4 . This local search is performed by solving the CP model $\mathcal{M}'(x)$ derived from \mathcal{M} by specifying that:

- The arrival time of j_0 is fixed, since $j_0 \notin I^*(x)$. This step is done by adding to \mathcal{M} the following constraint: $arrival_time(j_0, 0) = 0$. In practice, $arrival_time(j_0, 0)$ is declared in $\mathcal{M}'(x)$ as a constant with value 0.
- The arrival times of tasks in $I^*(x)$ are bounded within distance D from the arrival times computed by GA in x . For this example, suppose $D = 2$. This step is done by adding to \mathcal{M} the following constraints:

$$\begin{aligned} 2 - 2 = 0 &\leq arrival_time(j_1, 0) \leq 4 = 2 + 2 \\ 3 - 2 = 1 &\leq arrival_time(j_2, 0) \leq 5 = 3 + 2 \\ 6 - 2 = 4 &\leq arrival_time(j_3, 0) \leq 8 = 6 + 2 \\ 3 - 2 = 1 &\leq arrival_time(j_4, 0) \leq 5 = 3 + 2 \end{aligned}$$

In practice, $arrival_time(j_1)$ is declared in $\mathcal{M}'(x)$ as a variable with domain $[0, 4]$, $arrival_time(j_2)$ as a variable with domain $[1, 5]$, $arrival_time(j_3)$ as a variable with domain $[4, 8]$, and $arrival_time(j_4)$ as a variable with domain $[1, 5]$.

GA+CP solves this model to optimality and finds the solution $x' = [[0][2][3][4][3]]$, with the schedule shown in Figure 4.2. In x' , j_4 misses its deadline by 4 quanta, and therefore GA+CP has succeeded in improving the solution found by GA by finding a larger deadline miss.

6. EMPIRICAL STUDY

The goal of our empirical study is to compare the overall performance of GA, CP, and GA+CP for the purpose of supporting stress testing of task deadlines. Recall from Section 3 that approaches based on GA [Briand et al. 2006] and CP [Di Alesio et al. 2013] were proposed to support stress testing of task deadlines by searching for worst-case scenarios, and are therefore natural comparison baselines. To successfully enable our empirical study, we slightly modified the original GA approach as detailed in our previous work [Di Alesio et al. 2013]. Specifically, (1) we added support for multi-core platforms, as the original work was meant to apply only to software systems running on single-core architectures, and (2) we replaced the original fitness function [Briand et al. 2006] with the one in Equation (1) (Section 5.1) in order to account for deadline misses for all tasks, rather than for a single target task.

The comparison is performed on five subject systems reported in the literature, briefly described in Section 6.1. The goal of our study is to answer the research questions presented in Section 6.2 based on the metrics and attributes detailed in Section 6.3. The design of our experiment is described in Section 6.4, and its results are discussed in Section 6.5. Finally, Section 6.6 covers some potential threats that could affect the general validity of our conclusions.

6.1. Subject Systems

To investigate the general performance of GA, CP, and GA+CP in a variety of conditions, we selected five systems from safety-critical domains with varying size and complexity. These same systems have been selected for comparing GA and CP in our previous work [Di Alesio et al. 2013]. Specifically, our comparison is based on one system from the aerospace domain, two systems from the automotive domain, and two from the avionics domain. The subject systems 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 [Peraldi-Frati and Sorel 2008]. 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 [Anssi et al. 2011]. 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 [Traore et al. 2006]. 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 the IBM Federal Sector Division designed a specification for a hypothetical avionics software mission control computer of a military aircraft [Locke et al. 1990]. 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 of the two satellites Herschel and Planck [Mikučionis et al. 2010]. 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 2 summarizes relevant data from the systems' specifications, reported in ascending order of size and complexity. Specifically, we take into account the number of software tasks, interdependencies, triggering relations, and platform cores. This data has been extracted from the sources cited above, which include full descriptions of the systems. The complete version of the data extracted is available on-line as a technical report⁴.

	Software System				Platform	Logsize
	Tasks		Relationships		Cores	
	Periodic	Aperiodic	Dependencies	Triggering		
ICS	3	3	3	0	3	446.7
CCS	8	3	3	6	2	551.6
UAV	12	4	4	0	3	671.5
GAP	15	8	6	5	2	709.4
HPSS	23	9	5	0	1	836.6

Table 2: Subject systems data

To investigate the impact that the target system complexity has over the practical usefulness of the search strategies, we also quantified the system size. Specifically, we define the size of each system as the size of its associated search space, that is the product of the domain size of the search variables (Section 5.1). Indeed, the search space contains by definition all the feasible assignments of values to variables in the problem. The last column of the table reports the base 2 logarithm of the size of the search space. For example, in ICS there are circa $2^{446.7}$ possible ways in which tasks could arrive and be scheduled for execution.

⁴ <http://home.simula.no/~stefanod/data.pdf>

6.2. Research Questions

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

- **RQ1 — Efficiency.** Does one search strategy find the best solutions significantly faster than the other?
- **RQ2 — Effectiveness.** Does one search strategy find significantly better solutions (i.e., solutions with worse deadline misses) than the other?
- **RQ3 — Diversity.** Does one search strategy find solutions that are significantly more diverse (i.e., solutions that exercise the system in a larger number of different ways) than the other?
- **RQ4 — Scalability.** To what extent does the size of a system affect the efficiency of the three search strategies?

RQ1, RQ2 and RQ3 are investigated through a set of metrics and attributes detailed in Section 6.3. The goal of such metrics and attributes is to provide quantitative evidence to answer the research questions. RQ4 will instead be only qualitatively discussed in Section 6.5. This is because we base our analysis of efficiency on a set of five systems, and therefore no quantitative study, for example based on regression analysis, can be carried out to identify precise trends.

6.3. Comparison Metrics and Attributes

Though the search for optimal solutions is driven by function F defined in Equation (1) (Section 5.1), 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, effectiveness, and diversity properties related to RQ1, RQ2, and RQ3 as *attributes*, and we defined a set of *metrics* to enable their measurement. Therefore, we compare the performance of GA, CP and GA+CP by collecting data pertaining to the metrics and attributes defined below.

To enable a formal definition of metrics and attributes, we introduce the following notation:

- **Search Strategy Γ .** We denote the search strategies with the letter Γ : $\Gamma \in \{\text{GA}, \text{CP}, \text{GA+CP}\}$.
- **Subject System Σ .** We denote the systems described in Section 6.1 with the letter Σ : $\Sigma \in \{\text{ICS}, \text{CCS}, \text{UAV}, \text{GAP}, \text{HPSS}\}$.
- **Set of Solutions X .** We denote the set of solutions found by the search strategy Γ during an experiment on the target system Σ as $X(\Gamma, \Sigma)$.

6.3.1. Comparison Metrics. We introduce four main comparison metrics to capture four aspects of practical interest that characterize a stress test case.

- (1) The time needed to generate the test case.
- (2) The magnitude of deadline misses identified by the test case.
- (3) The number of tasks identified to miss a deadline.
- (4) The number of task executions identified to miss a deadline.

These four properties are formalized through the following four metrics. Each metric is defined for a given solution $x \in X$ found by the search strategy Γ during an experiment on the subject system Σ . In the definitions, we omit the dependency from Γ and Σ for the sake of readability. Recall from Section 5.2 that a solution x is defined as a sequence of arrival times $x_{j,k}$ for each aperiodic task, i.e., $x = [[x_{j,k} \mid k \in K_j] \mid j \in J_a]$.

- **Computation time t .** We define $t(x)$ as the time required to find solution x , from when the search starts.
- **Sum s of time quanta in deadline misses.** 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 into 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. We define $s(x)$ as the sum of time quanta in all deadline misses of solution x . Recall from Section 5.1 that, for a given solution x , we define $deadline_miss(j,k) \stackrel{\text{def}}{=} end(j,k) -$

$task_deadline(j,k)$. Therefore, we define $s(x)$ in the following way:

$$s(x) \stackrel{\text{def}}{=} \sum_{j \in J, k \in K_j(x)} \max(0, deadline_miss_{j,k}(x))$$

- **Number n of tasks that miss a deadline.** This number is relevant for generating stress test cases for task deadlines as, in practice, every task that misses a deadline has to be looked into and possibly re-designed. Hence, not realizing that a task can miss its deadline may lead to overlooking an important flaw.

We define $n(x)$ as the number of tasks that miss at least a deadline in solution x :

$$n(x) \stackrel{\text{def}}{=} |\{j \in J \mid \exists k \in K_j(x) \cdot deadline_miss_{j,k}(x) \geq 0\}|$$

- **Number m of task executions that miss a deadline.** This number is also of interest as, in soft real-time systems, one could tolerate less critical tasks missing some deadlines, provided that the frequency of deadline misses is acceptable. Therefore, overestimating m might lead us to inspect a task when unnecessary, and underestimating m could lead to overlooking tasks that frequently miss their deadlines. Note that, by definition, $\forall x \in X \cdot m(x) \geq n(x)$.

We define $m(x)$ as the number of task executions that miss a deadline in solution x :

$$m(x) \stackrel{\text{def}}{=} |\{k \in K_j(x) \mid j \in J \wedge deadline_miss_{j,k}(x) \geq 0\}|$$

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. Specifically, solutions with many large deadline misses or many tasks or task executions that miss a deadline characterize worst case scenarios. Therefore, a *best* solution can be identified only with respect to a specific metric.

To enable the formal definition of the comparison attributes in Section 6.3.2, we also define the following quantities for each search strategy Γ running during an experiment on the target system Σ :

- **Largest sum s^* of time quanta in deadline misses.** We define s^* as the value for the largest deadline miss in X , i.e., $s^* \stackrel{\text{def}}{=} \max_{x \in X} s(x)$.
- **Largest number n^* of tasks missing their deadline.** We define n^* as the largest number of tasks that miss their deadline in X , i.e., $n^* \stackrel{\text{def}}{=} \max_{x \in X} n(x)$.
- **Largest number m^* of task executions missing their deadline.** We define m^* as the largest number of task executions that miss their deadline in X , i.e., $m^* \stackrel{\text{def}}{=} \max_{x \in X} m(x)$.
- **Set X_s^* of best solutions with respect to s .** We define X_s^* as the set of solutions that have the largest sum s^* of time quanta in deadline misses, i.e., $X_s^* \stackrel{\text{def}}{=} \{x \in X \mid s(x) = s^*\}$.
- **Set X_n^* of best solutions with respect to n .** We define X_n^* as the set of solutions that have the largest number n^* of tasks missing at least one deadline, i.e., $X_n^* \stackrel{\text{def}}{=} \{x \in X \mid n(x) = n^*\}$.
- **Set X_m^* of best solutions with respect to m .** We define X_m^* as the set of solutions that have the largest number m^* of task executions missing a deadline, i.e., $X_m^* \stackrel{\text{def}}{=} \{x \in X \mid m(x) = m^*\}$.

6.3.2. Comparison Attributes. We introduce three main comparison attributes to capture three aspects of practical interest while testing. Indeed, an ideal test suite has three main properties:

- (1) It is computed in the shortest possible time.
- (2) It contains test cases that are as likely as possible to push tasks to miss their deadlines at runtime.
- (3) It contains test cases that are as little redundant as possible.

Efficiency captures the first property, measuring how quickly a search strategy converges to the optimal solutions it finds. *Effectiveness* captures the second property, measuring how likely are the solutions found to characterize stress cases that will reveal deadline misses at runtime. Finally, the third property is captured by the concept of *diversity*. Conceptually, diversity among stress test cases

is similar to test coverage in functional testing. Indeed, a test suite that yields high coverage with respect to a given criterion ensures that the system will be thoroughly tested with respect to that criterion. Similarly, a test suite that yields high diversity ensures that test cases will thoroughly exercise interactions between task executions.

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

- **Efficiency** η . The more efficient a strategy, the faster it computes its best solutions. Therefore, the efficiency attribute relates to RQ1 and RQ4.

We define the efficiency η with respect to a given metric as the minimum time required to compute one of the best solutions with respect to that metric. Specifically, we define the efficiency with respect to s , m , and n as:

$$\eta_s \stackrel{\text{def}}{=} \min_{x \in X_s^*} t(x) \quad \eta_n \stackrel{\text{def}}{=} \min_{x \in X_n^*} t(x) \quad \eta_m \stackrel{\text{def}}{=} \min_{x \in X_m^*} t(x)$$

- **Effectiveness** κ . The more effective a strategy, the better the solutions it computes. Therefore, the efficiency attribute relates to RQ2.

We define the effectiveness κ with respect to a given metric as the value of that metric for the best solutions found. Specifically, we define effectiveness with respect to s , m , and n as:

$$\kappa_s \stackrel{\text{def}}{=} s^* \quad \kappa_n \stackrel{\text{def}}{=} n^* \quad \kappa_m \stackrel{\text{def}}{=} m^*$$

For example, in Figure 4.1 $\kappa_s = 2$ and in Figure 4.2 $\kappa_s = 4$.

- **Number N of best solutions**. In general, the higher the number of best solutions of a search strategy, the higher the number of effective test cases it generates. However, the effectiveness κ is a different concept from the number N of solutions which are effective. Since N does not take into account redundancy among solutions, it is not true in general that the higher N is, the more effective a search strategy is. For example, a strategy could generate a large number of effective test cases that yet exercise the system with respect to very similar scenarios. On the other hand, another strategy could generate fewer effective test cases that are instead highly diverse. For this reason, the number of best solutions is meaningful only when considered together with the redundancy of the solutions, formalized by the concept of diversity. Therefore, the number of best solutions relates to RQ3.

We define the number N of best solutions with respect to a given metric as the cardinality of the set of best solutions found with respect to that metric. Specifically, we define effectiveness with respect to s , m , and n as:

$$N_s \stackrel{\text{def}}{=} |X_s^*| \quad N_n \stackrel{\text{def}}{=} |X_n^*| \quad N_m \stackrel{\text{def}}{=} |X_m^*|$$

- **Diversity** δ . As explained before, a search strategy may find redundant solutions which stress the system under similar task schedules. However, covering a diverse set of interactions between the tasks executions is important not to overlook potentially faulty schedules. To capture similarities and differences between the schedules produced by solutions, we define three types of diversity δ , each with respect to a given metric. Specifically, we define the shift diversity δ_h , the pattern diversity δ_r , and the execution diversity δ_e , each defined with respect to the three metrics s , m , and n . These diversity attributes also relate to RQ3. Intuitively, the shift diversity δ_h measures the extent to which solutions exercise the system during time intervals that are distant from each other. The shift diversity is defined based on the shift distance between active vectors, which measures the distance in time between a given execution in two solutions.

- **Shift distance \mathcal{D}_h between active vectors**. Let $A_{j,k}(x)$ and $A_{j,k}(y)$ be the active vectors of task execution (j,k) in the solutions x and y , respectively.

We define the shift distance $\mathcal{D}_{h,j,k}(x,y)$ between $A_{j,k}(x)$ and $A_{j,k}(y)$ as the sum of absolute differences between their start and end times:

$$\mathcal{D}_{h,j,k}(x,y) \stackrel{\text{def}}{=} |\text{start}_{j,k}(x) - \text{start}_{j,k}(y)| + |\text{end}_{j,k}(x) - \text{end}_{j,k}(y)|$$

For example, in Figures 4.1 and 4.2, $\mathcal{D}_{h,j_1,0} = |2 - 2| + |6 - 8| = 2$.

Similarly, we define the shift distance $\mathcal{D}_{h,j}(x,y)$ between $A_j(x)$ and $A_j(y)$ as the average shift distance over pairs of executions of task j in solutions x and y :

$$\mathcal{D}_{h,j}(x,y) \stackrel{\text{def}}{=} \frac{\sum_{k \in (K_j(x) \cap K_j(y))} \mathcal{D}_{h,j,k}(x,y)}{|K_j(x) \cap K_j(y)|}$$

In Figures 4.1 and 4.2, $\mathcal{D}_{h,j_1} = \mathcal{D}_{h,j_1,0} = 2$, since j_1 is executed only once.

- **Shift Diversity δ_h .** We define the shift diversity $\delta_h(A(x), A(y))$ between solutions x and y as the sum of the shift distances between $A_j(x)$ and $A_j(y)$ for $j \in J$:

$$\delta_h(A(x), A(y)) \stackrel{\text{def}}{=} \sum_{j \in J} \mathcal{D}_{h,j}(x,y)$$

Similarly, we define the shift diversity δ_h with respect to a given metric as the average shift diversity over the set of best solutions for that metric:

$$\delta_{h,s} \stackrel{\text{def}}{=} \frac{\sum_{x,y \in X_s^*} \delta_h(x,y)}{|X_s^*|} \quad \delta_{h,n} \stackrel{\text{def}}{=} \frac{\sum_{x,y \in X_n^*} \delta_h(x,y)}{|X_n^*|} \quad \delta_{h,m} \stackrel{\text{def}}{=} \frac{\sum_{x,y \in X_m^*} \delta_h(x,y)}{|X_m^*|}$$

In Figures 4.1 and 4.2, $\delta_h = 0 + 2 + 2 + 8 + 4 = 16$.

The pattern diversity δ_r measures the extent to which solutions exercise the system such that tasks are preempted at different times. The pattern diversity is defined based on the pattern distance between active vectors, which measures the difference between the preemption times of a given task execution in two solutions.

- **Pattern distance \mathcal{D}_r between active vectors.** Let $A_{j,k}(x)$ and $A_{j,k}(y)$ be the active vectors of task execution (j,k) in the solutions x and y , respectively. We define the pattern distance $\mathcal{D}_{r,j,k}(x,y)$ between $A_{j,k}(x)$ and $A_{j,k}(y)$ as the sum of the absolute differences between the preemption values of task j :

$$\mathcal{D}_{r,j,k}(x,y) \stackrel{\text{def}}{=} \sum_{p \in P_j \setminus \{0\}} |\text{preempted}_{j,k,p}(x) - \text{preempted}_{j,k,p}(y)|$$

For example, in Figures 4.1 and 4.2, $\mathcal{D}_{r,j_2,0} = |0 - 2| = 2$.

Similarly, we define the pattern distance $\mathcal{D}_{r,j}(x,y)$ between $A_j(x)$ and $A_j(y)$ as the average shift distance over pairs of executions of task j in solutions x and y :

$$\mathcal{D}_{r,j}(x,y) \stackrel{\text{def}}{=} \frac{\sum_{k \in (K_j(x) \cap K_j(y))} \mathcal{D}_{r,j,k}(x,y)}{|K_j(x) \cap K_j(y)|}$$

In Figures 4.1 and 4.2, $\mathcal{D}_{r,j_2} = \mathcal{D}_{r,j_2,0} = 2$, since j_2 is executed only once.

- **Pattern Diversity δ_r .** We define the diversity $\delta_r(A(x), A(y))$ between the solutions x and y as the sum of the pattern distances between $A_j(x)$ and $A_j(y)$ for $j \in J$:

$$\delta_r(A(x), A(y)) \stackrel{\text{def}}{=} \sum_{j \in J} \mathcal{D}_{r,j}(x,y)$$

Similarly, we define the pattern diversity δ_r with respect to a given metric as the average pattern diversity over the set of best solutions for that metric:

$$\delta_{r,s} \stackrel{\text{def}}{=} \frac{\sum_{x,y \in X_s^*} \delta_r(x,y)}{|X_s^*|} \quad \delta_{r,n} \stackrel{\text{def}}{=} \frac{\sum_{x,y \in X_n^*} \delta_r(x,y)}{|X_n^*|} \quad \delta_{r,m} \stackrel{\text{def}}{=} \frac{\sum_{x,y \in X_m^*} \delta_r(x,y)}{|X_m^*|}$$

In Figures 4.1 and 4.2, $\delta_r = 0 + 2 + 2 + 0 + 0 = 4$.

The execution diversity δ_e measures the extent to which solutions exercise the system such that tasks are executed different numbers of times.

- **Execution Diversity δ_e .** We define the execution diversity $\delta_e(A(x), A(y))$ between the solutions x and y as the sum of the absolute differences between the number of task executions in solutions x and y :

$$\delta_e(A(x), A(y)) \stackrel{\text{def}}{=} \sum_{j \in J} \left| |K_j(x)| - |K_j(y)| \right|$$

Similarly, we define the execution diversity δ_h with respect to a given metric as the average execution diversity over the set of best solutions for that metric:

$$\delta_{e,s} \stackrel{\text{def}}{=} \frac{\sum_{x,y \in X_s^*} \delta_e(x,y)}{|X_s^*|} \quad \delta_{e,n} \stackrel{\text{def}}{=} \frac{\sum_{x,y \in X_n^*} \delta_e(x,y)}{|X_n^*|} \quad \delta_{e,m} \stackrel{\text{def}}{=} \frac{\sum_{x,y \in X_m^*} \delta_e(x,y)}{|X_m^*|}$$

For example, in Figures 4.1 and 4.2, $\delta_e = |1 - 1| + |1 - 1| + |1 - 1| + |1 - 1| + |1 - 1| = 0$ because each task gets executed once in each solution.

Diversity Properties. We note that δ_h , δ_r , and δ_e are defined as non-negative, symmetric, and subadditive. Furthermore, when considered in conjunction, the three diversities also satisfy the coincidence property. Specifically, δ_h , δ_r , and δ_e satisfy the following four properties:

- (1) **Non-Negativity**

$$\forall x, y \cdot \delta_h(x, y) \geq 0 \wedge \delta_r(x, y) \geq 0 \wedge \delta_e(x, y) \geq 0$$

- (2) **Coincidence**

$$\forall x, y \cdot \delta_h(x, y) = 0 \wedge \delta_r(x, y) = 0 \wedge \delta_e(x, y) = 0 \iff x = y$$

- (3) **Symmetry**

$$\forall x, y \cdot \delta_h(x, y) = \delta_h(y, x) \wedge \delta_r(x, y) = \delta_r(y, x) \wedge \delta_e(x, y) = \delta_e(y, x)$$

- (4) **Subadditivity (Triangle inequality)**

$$\begin{aligned} \forall x, y, z \cdot \delta_h(x, y) + \delta_h(y, z) &\geq \delta_h(x, z) \wedge \\ \delta_r(x, y) + \delta_r(y, z) &\geq \delta_r(x, z) \wedge \\ \delta_e(x, y) + \delta_e(y, z) &\geq \delta_e(x, z) \end{aligned}$$

The proofs of the above four properties are straightforward and follow from the definition of δ_h , δ_r , and δ_e as sums of absolute values. These properties enable the definition of the three types of diversity as distance functions on the set of solutions.

Diversity Examples. In this section, we present an example with three pairs of solutions x and y . Each pair represents a case where only one type of diversity, δ_h , δ_r , and δ_e respectively, has a non-zero value. This highlights the necessity of breaking down the concept of diversity into three orthogonal sub-attributes that have to be considered together when analyzing how differently stress test cases exercise the system.

Consider the single-task system detailed in Table 3. Note that *period* and *offset* are not defined, since j_0 is aperiodic. Trivially, there are also no *dependency* or *triggers* relationships.

Figure 5 shows six different solutions for the system detailed in Table 3. In their schedules, we omitted the arrival times and deadlines of j_0 , since they are not relevant for the definition of diversity. Note that j_0 is the only task of the system, and therefore it cannot be preempted. However, in these example solutions, we introduce unnecessary task preemptions to meaningfully describe the concept of diversity.

Consider the two solutions x_1 and y_1 shown in Figures 5.1 and 5.2. In this case, we note that $\delta_h(x_1, y_1) = |2 - 0| + |9 - 7| = 4$. This reflects the fact that x_1 is predicted to exercise the system

	priority	duration	min_ia	max_ia	deadline
j_0	0	3	4	10	6
$T = [0, 9]$			$c = 1$		

Table 3: Example system with one task

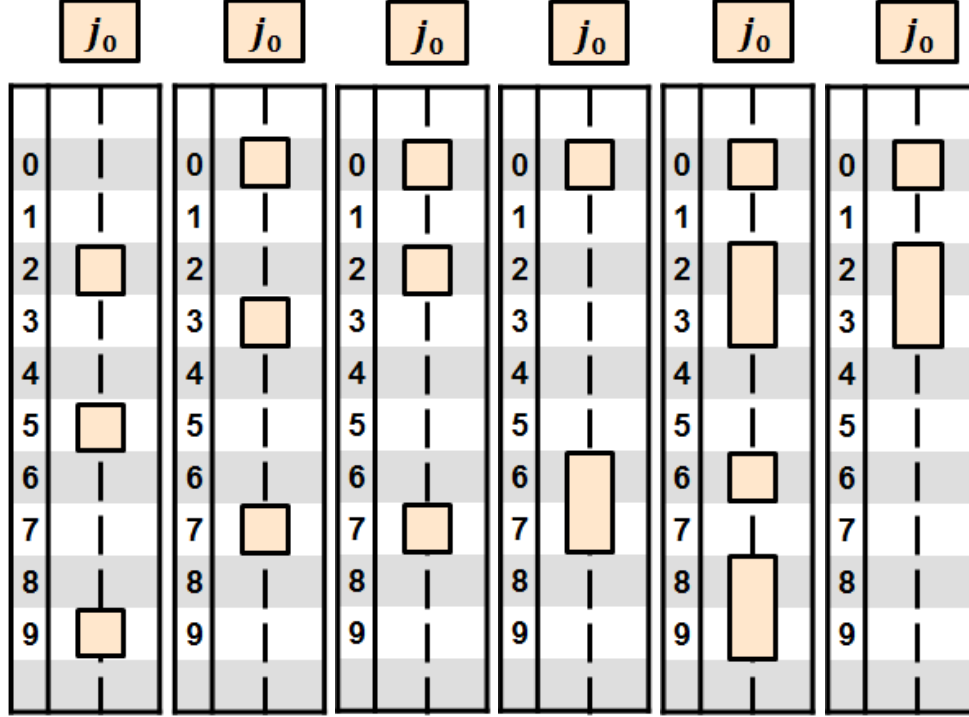
(1) Solution x_1 (2) Solution y_1 (3) Solution x_2 (4) Solution y_2 (5) Solution x_3 (6) Solution y_3

Fig. 5: Example solutions for the system in Table 3

in the interval $[2, 9]$, and y_1 is predicted to do so in $[0, 7]$. Furthermore, $\delta_r(x_1, y_1) = |3 - 3| + |4 - 4| = 0$. This reflects the fact that in both x_1 and y_2 task j_0 is predicted to preempt at runtime in the same way, i.e., by two time quanta the first time, and by three the second time. Finally, $\delta_e(x_1, y_1) = |1 - 1| = 0$ since in both solutions j_0 gets executed once.

Furthermore, consider the two solutions x_2 and y_2 in Figures 5.3 and 5.4. In this case, we note that $\delta_h(x_2, y_2) = |0 - 0| + |7 - 7| = 0$. This reflects the fact that x_2 and y_2 are both predicted to exercise the system in the same interval $[0, 7]$. However, $\delta_r(x_2, y_2) = |1 - 5| + |4 - 0| = 8$. This reflects the fact that in x_2 and y_2 task j_0 is predicted to preempt at runtime in different ways. Indeed, j_0 is preempted twice for 1 and 4 time quanta in x_3 , while in y_2 it is preempted once for 5 time quanta. Finally, $\delta_e(x_2, y_2) = |1 - 1| = 0$ since even in this case j_0 gets executed once in both solutions.

Finally, consider the two solutions x_3 and y_3 in Figures 5.5 and 5.6. In this case, we note that $\delta_h(x_3, y_3) = |0 - 0| + |3 - 3| = 0$. This reflects the fact that, considering only the first execution of j_0 that is present in both solutions, x_3 and y_3 are predicted to exercise the system in the same interval $[0, 4]$. Moreover, $\delta_r(x_3, y_3) = |1 - 1| + |0 - 0| = 0$. This reflects the fact that, considering again the only common execution of j_0 , x_3 and y_3 are both predicted to preempt j_0 once by 1 time quanta. However, $\delta_e(x_3, y_3) = |2 - 1| = 1$ since in this case j_0 gets executed twice in x_3 and only once in y_3 .

6.4. Experiment Set-Up

To answer the research questions, we performed a series of experiments over the systems described in Section 6.1. The experimental design is illustrated in Figure 6. Each experiment consisted of running GA, CP and GA+CP on a target system Σ for a number of times, each run generating a set X of solutions. For GA, we used the implementation introduced by Briand et al. [Briand et al. 2006], while for CP we used the constrained optimization model defined in our previous work [Di Alesio et al. 2014]. Since the purpose of our empirical study is to compare the practical usefulness of the three search strategies, we chose to run them 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 GA and CP for ten hours. To do so, we set up GA to continuously generate new solutions for ten hours, while we set up CP to terminate the search after ten hours. GA+CP was instead run by performing one local CP search for each solution found by GA. We set a timeout of two hours for these local searches, so that GA+CP was run for a total of 12 hours. However, note that CP terminates the search upon proof of optimality. Since CP performed the local searches in a significantly small subset of the search space (recall Figure 3), the local CP searches always terminated with proof of optimality before the 2 hours timeout. Therefore, even if we instructed GA+CP to run longer than GA and CP, the time taken by CP to terminate the local searches was practically not significant with respect to the 10 hours taken by GA to generate its solutions. For each run of GA, CP, and GA+CP, we recorded in X only the 100 solutions with the highest fitness/objective value found. This is because each solution characterizes a stress test case, and 100 has proven to be a satisfactory number of observations to meaningfully compare two distributions [Arcuri and Briand 2011]. Similar to GA and CP, we instructed GA+CP to run 100 CP searches as described in Section 5.2, each in the neighborhood of a solution found by GA.

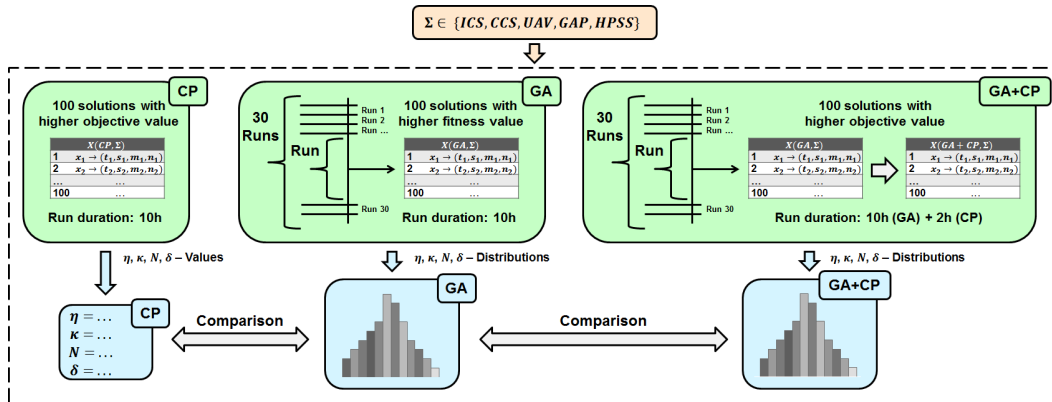


Fig. 6: Experimental design: we run CP a single time recording the 100 solutions with highest objective value, and calculating a single value for each metric. Then, we run GA and GA+CP 30 times recording the 100 solutions with highest fitness value, and calculating distributions for the metrics.

Running the three search strategies for approximately the same amount of time allows us to meaningfully compare effectiveness, number of solutions found, and diversity. Furthermore, during the design of the experiment, we had to consider the inherent randomized behavior of GA in contrast to the full determinism 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 each system, we ran GA, and consequently GA+CP 30 times on each system. In this way, we could compute the comparison metrics distributions of the best solutions recorded

over 30 runs. Since our research questions are directly related to attributes η , κ , N , and δ , for each solution $x \in X$ we computed the values of the metrics t , s , n , and m used to define such attributes. GA, CP and GA+CP runs have been separately executed on a single Amazon EC2 m2.xlarge instance⁵.

6.5. Results and Discussion

In this section, we discuss the experimental results for the attributes η , κ , N , and δ for each subject system. Each attribute is discussed through 15 box-and-whisker plots, one for each pair of metric and system. In each plot, the x-axis reports the search strategies CP, GA, and GA+CP, respectively denoted as C, G, and +. The y-axis reports instead the value for the comparison attribute. Each plot is also complemented by two p-values from the non-parametric Wilcoxon statistical significance test between GA+CP and CP (first row), and between GA+CP and GA (second row). Specifically, we report the p-values from the Wilcoxon rank-sum test for the difference between GA+CP and GA, and the Wilcoxon signed-rank test for the difference between GA+CP and CP. The former is a two-sample test comparing two distributions, while the latter is a one-sample test, given the deterministic nature of CP. In particular, we investigated the statistical significance of differences between CP and GA/GA+CP by testing the null hypothesis that the median of the GA/GA+CP distributions are the deterministic values obtained with CP. For all tests, we selected a level of significance $\alpha = 0.05$. Note that a centered dot (\cdot) in place of a Wilcoxon test p-value indicates that the test has not been performed. This happens when the two distributions considered for the test are identical, and hence the effective sample size, i.e., the number of observation pairs with different values, is zero.

6.5.1. RQ1 — Efficiency. The first three columns in Table 4 report the efficiency η with respect to s , n and m for GA+CP, GA, and CP, and for each subject system. The computation times for the best solutions are reported in the format *hh:mm*. We observe that, on each system, GA+CP has a worse efficiency than GA with respect to each metric. With the exclusion of η_n for ICS and UAV, the difference in efficiency is also statistically significant as shown by the p-values below 0.05. This result is expected because GA+CP performs a complete search in the neighborhood of GA solutions, and therefore, the time GA+CP requires to find its best solution is in general higher than that of GA. However, the difference between the average efficiency of GA+CP and GA is small from a practical standpoint, and does not keep GA+CP from being far more efficient than CP. Specifically, the difference between the η_s , η_n and η_m medians of GA and GA+CP vary from around 20 minutes in ICS (Tables 4.1 to 4.3) up to 1.5 hour in HPSS (Tables 4.37 to 4.39). This difference has little practical significance when compared to the ten hours duration of each run, and to the efficiency of CP, that varies from 3.5 hours in ICS up to almost ten hours in HPSS. The statistical significance of the difference between the efficiency of GA+CP and CP is also reflected in the p-value for the signed-rank test, which is below 0.0001 for each metric and subject system. On average, the results show that GA+CP is twice as fast than CP but only 20% slower than GA in finding the best solutions x_s^* , x_n^* , and x_m^* .

6.5.2. RQ2 — Effectiveness. The second three columns in Table 4 reports the effectiveness κ with respect to s , n and m for GA+CP, GA, and CP, and for each subject system. We observe how, on each system, GA+CP has equal or greater effectiveness than that of GA with respect to each metric. This result is also expected, since GA+CP performs a complete search in the neighborhood of GA solutions, and thus the solutions it finds are always equal or better than those of GA. We note how GA+CP is significantly more effective than GA for two out of three metrics in most of the subject systems, being so for all three metrics in GAP and HPSS. Moreover, for n and m , GA+CP also achieves nearly the same effectiveness as CP in most of the subject systems. In particular, in ICS, CCS and UAV, GA+CP achieves the same value as CP for κ_n in the first quartile (Tables 4.5, 4.14 and 4.23), and for κ_m in the second quartile (Tables 4.6, 4.15 and 4.24). On the other hand, for these three subject systems, GA achieves the same effectiveness as CP on the second or third quartile on average. Note that, in CCS and UAV, GA never achieves the same effectiveness as CP for the criteria

⁵ <http://aws.amazon.com/>

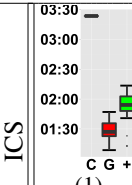
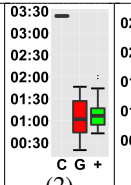
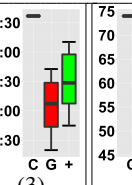
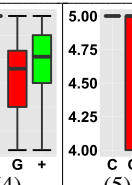
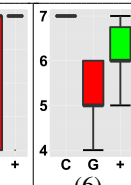
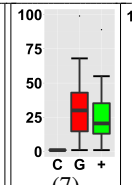
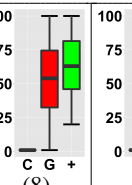
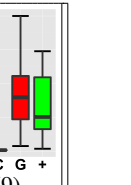

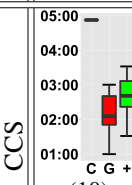
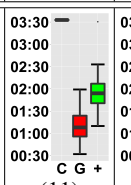
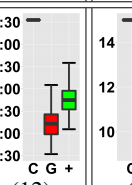
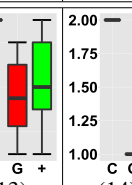
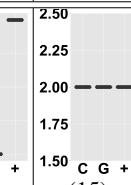
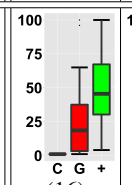
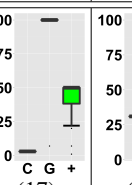
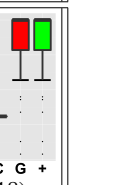

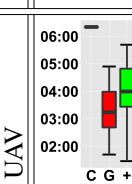
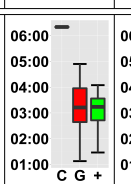
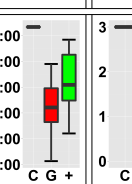
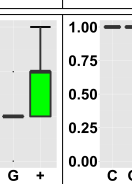
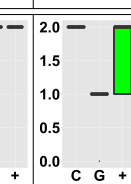
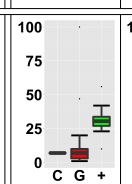
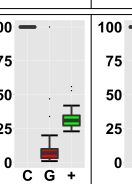
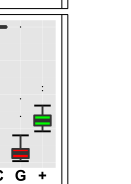

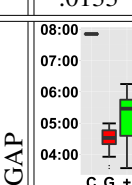
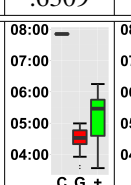
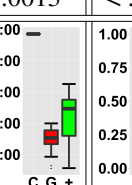
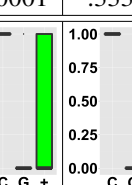

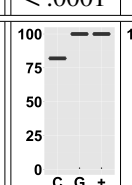
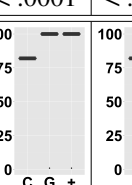


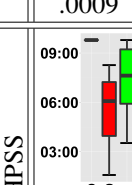
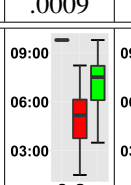
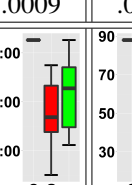
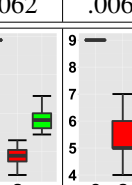
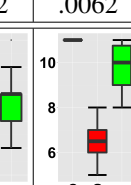
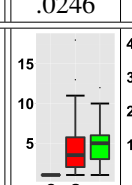
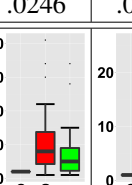
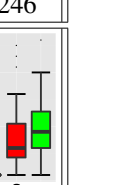

	η_s	η_n	η_m	κ_s	κ_n	κ_m	N_s	N_n	N_m
ICS									
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
	< .0001 < .0001	< .0001 .3403	< .0001 .0026	< .0001 .0730	< .3256 .0062	< .0001 < .0001	< .0001 .2640	< .0001 .1932	< .0001 .1784
CCS									
	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
	< .0001 < .0016	< .0001 .0001	< .0001 .0001	< .0001 .1996	. .0001	. .	< .0001 .1996	< .0001 .0001	< .0001 .
UAV									
	(19)	(20)	(21)	(22)	(23)	(24)	(25)	(26)	(27)
	< .0001 .0133	< .0001 .6309	< .0001 .0013	< .0001 < .0001	. .3337	.0005 < .0001	< .0001 < .0001	< .0001 < .0001	< .0001 < .0001
GAP									
	(28)	(29)	(30)	(31)	(32)	(33)	(34)	(35)	(36)
	< .0001 .0009	< .0001 .0009	< .0001 .0009	< .0001 .0062	< .0001 .0062	< .0001 .0062	.5171 .0246	.5171 .0246	.5171 .0246
HPSS									
	(37)	(38)	(39)	(40)	(41)	(42)	(43)	(44)	(45)
	< .0001 .0015	< .0001 .0003	< .0001 .0420	< .0001 < .0001	< .0001 < .0001	< .0001 < .0001	< .0001 .2215	.0004 .0605	< .0001 .1324

Table 4: Experimental results of CP (C), GA (G), and GA+CP (+) for efficiency η , effectiveness κ , and number N of best solutions. Each box-and-whisker plot reports at the bottom the Wilcoxon test p-values between GA+CP and CP (first value), and between GA+CP and GA (second value).

n and m . In GAP, GA+CP achieves the same effectiveness as CP in the third quartile for all three criteria, identifying a deadline miss in one third of the runs (Tables 4.31 to 4.33). In comparison, GA found a deadline miss only in a single run. Finally, in HPSS, GA+CP does not match the effectiveness of CP, but significantly improves the results of GA, having a first quartile that is around 30% larger than that of GA for s and m (Tables 4.40 and 4.42). Finally, we note that, with the exception of κ_s for ICS and CCS, whenever there is a statistically significant, positive difference in the effectiveness between CP and GA+CP, there is also one between GA+CP and GA. On average, the results show that GA+CP is significantly more effective than GA, and is approaching CP in finding the best solutions x_n^* and x_m^* .

6.5.3. *RQ3 — Diversity.* The last three columns in Table 4 report the number N of best solutions with respect to s , n and m for GA+CP, GA, and CP, and for each subject system. We observe how, on most systems, the number of best solutions found by GA+CP is similar to that of GA, even though it is not consistently larger or smaller. We conjecture that the reason for this result stems from the way GA+CP is designed to improve the solutions found by GA. Consider the scenarios shown in Figure 7.

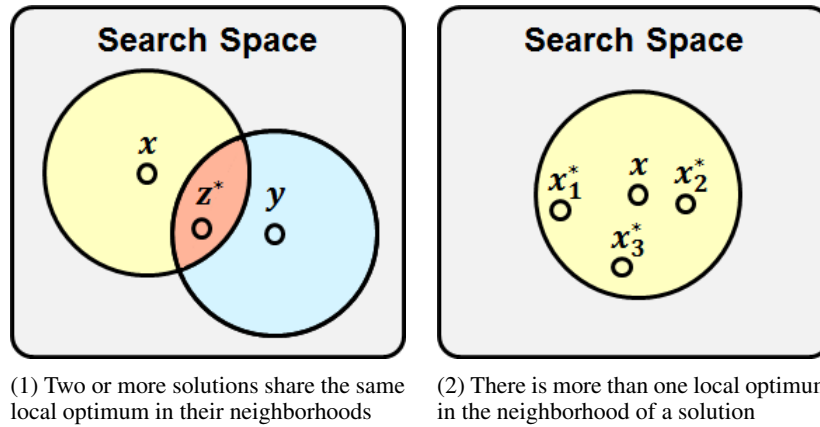


Fig. 7: Different scenarios of how GA+CP affects the number N of best solutions

Suppose that GA finds two distinct best solutions with respect to a metric, namely x and y . It could be the case that the best solution in the neighborhoods of x and y is the solution z^* . In this case, the number N of best solutions found by GA+CP is smaller than that of GA, because $N(\text{GA}) = 2$, and $N(\text{GA+CP}) = 1$ (Figure 7.1). Suppose now that GA finds only a single best solution x with respect to a metric. It could be the case instead, that there is more than a single best solution in the neighborhood of x , namely x_1^* , x_2^* , and x_3^* . In this scenario, the number N of best solutions found by GA+CP is larger than that of GA, because $N(\text{GA}) = 1$, and $N(\text{GA+CP}) = 3$ (Figure 7.2). In general, the two scenarios can happen independently from other factors since they depend only on the specific solutions found by GA in a subject system. For this reason, there is no clear trend on whether GA+CP finds a larger or smaller number of best solutions than GA with respect to a metric, as shown by the p-values. We note that, with the exception of GAP and UAV for N_n and N_m (Tables 4.26 and 4.27), GA and GA+CP find a significantly larger number of best solutions than CP. Note that in ICS, CP finds only a single best solution with respect to s , n , and m (Tables 4.7 to 4.9), and does so for s and m in HPSS (Tables 4.43 and 4.45), and for s in CCS (Table 4.16). This result is expected because of the randomized nature of GA, and consequently of GA+CP. Indeed, GA finds its solutions by starting from a randomly selected initial population of 80 individuals [Briand et al. 2006] and in our experiment we set up GA to continuously generate and evaluate solutions for ten hours. Even in cases where no deadline misses are revealed, GA is likely to generate a large

set of solutions from the initial population. On the other hand, CP is designed to find solutions from scratch with a *branch-and-bound* search process that progressively assigns values to variables in order to satisfy constraints [Apt 2003]. Furthermore, CP discards by design solutions that have worse objective values than the current best known solution [Atamtürk and Savelsbergh 2005], and is thus less likely to generate a large set of solutions. Recall that, in our analysis, each solution characterizes a stress test case. Therefore, a large number of solutions is indicative of the size of test suite generated by the search strategies. However, we note that N itself is not sufficient by itself to give a practical measure for the test suite dimension, because many of the solutions found could be redundant (Section 6.3). For this reason, the number N of best solutions needs to be considered together with their diversity.

Table 5 reports the diversity δ_h , δ_r , and δ_e with respect to s , n and m for GA+CP, GA, and CP, and for each subject system. We observe how, on each system, the three diversities of GA+CP are similar to those of GA, even though they are not consistently larger or smaller. We conjecture that the reason why GA+CP retains a number N of best solutions similar to GA also explains this result. This means that the local search performed by CP in the neighborhood of solutions computed by GA had no significant effect over the three types of diversity in our experiment. Therefore, in our subject systems, the solutions found by GA+CP retained a diversity similar to that of GA. Note that, as expected, in the cases where CP found only a single best solution with respect to a metric, the three diversities have a null value. In GAP, CP found 82 best solutions with respect to s , n , and m , as opposed to 100 for GA and GA+CP. However, the solutions found by GA and GA+CP are far less redundant than those of CP, having a significantly higher shift, pattern, and execution diversity (Tables 5.28 to 5.36). The same also holds for criterion s in UAV (Tables 5.19, 5.22 and 5.25), where the three search strategies found 100 best solutions. We finally remark that the diversity is not normalized, and the values are meant to be compared only within the same subject system and the same type of diversity. Recall from Section 6.3 that δ_h is defined in terms of start and end times of task executions, and depends mostly on the observation interval T . δ_r is defined in terms of preemptions between task executions, and depends mostly on the task durations. Finally, δ_e is defined in terms of number of task executions, and depends mostly on the ratio between T and the minimum and maximum inter-arrival times of aperiodic tasks. Since T is usually much larger than task durations, δ_h has higher values than δ_r and δ_e . For example, in ICS, the average value for $\delta_{h,s}$ (GA+CP) is 83.43. This means that on average, the task executions in the best solutions with respect to s found by GA+CP are shifted by 83.43 time quanta. Similarly, in ICS the average for $\delta_{r,s}$ (GA+CP) is 41.08, meaning that the preemptions between task executions in the best solutions with respect to s found by GA+CP differ on average by 41.08 time quanta. Finally, in ICS the average for $\delta_{e,s}$ (GA+CP) is 1.56, meaning that on average the tasks in the best solutions with respect to s differ by 1.56 executions.

6.5.4. R4 — Scalability. Figure 8 reports the trend of the efficiency η with respect to the system size for GA+CP (downward triangle), GA (full dot), and CP (diamond). In each graph, the x-axis represents the system size, while the y-axis reports the efficiency of the search strategies. Since we have a set of five subject systems, we did not perform any statistical analysis, and we only limit ourselves to a qualitative discussion. However, for the three criteria the efficiency of GA+CP seems to scale linearly with system size. Indeed, since we represent the size as a logarithm (Section 6.1), the x-axis in the three graphs has a logarithmic scale, while the y-axis has the usual linear scale. For this reason, the apparently exponential shape of the efficiency trend is in practice linear. Such results are encouraging as they suggest that GA+CP is more to scale to large systems, and therefore be an advantageous solution given its practical trade-off between efficiency, effectiveness and diversity. In particular, note that, even if in HPSS the difference in efficiency between GA+CP and GA is larger than in the other subject systems, to this decrease in efficiency corresponds a significant increase in effectiveness, as discussed before. Finally, we remark that we did not investigate the trend for κ , N , and δ with respect to system size. This is because, unlike η , these last three properties depend primarily on the specific problem being solved, rather than on its size. For example, there could be


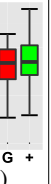
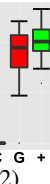
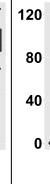

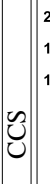
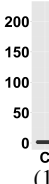
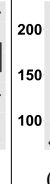

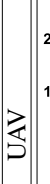



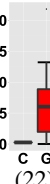
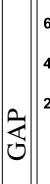

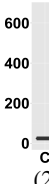
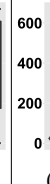


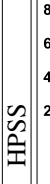


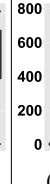

	$\delta_{h,s}$	$\delta_{h,n}$	$\delta_{h,m}$	$\delta_{r,s}$	$\delta_{r,n}$	$\delta_{r,m}$	$\delta_{e,s}$	$\delta_{e,n}$	$\delta_{e,m}$
ICS									
	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001
	.4289	.0406	.5493	.7958	.1154	.9705	.9176	.4688	.8650
CCS									
	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001
	.0070	.7394	.7283	.0536	.0948	.1297	.0003	.9352	1
UAV									
	< .0001	.0761	.0761	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001
	.2061	.1087	.1087	< .0001	.0002	.0002	< .0001	< .0001	< .0001
GAP									
	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001	< .0001
	.0021	.0021	.0021	.0034	.0034	.0034	.0913	.0913	.0913
HPSS									
	< .0001	< .0001	< .0001	< .0001	< .0001	.0004	< .0001	< .0001	< .0001
	.8071	.3147	.0004	.9410	.4597	.5692	.0075	.7061	.0314

Table 5: Experimental results of CP (C), GA (G), and GA+CP (+) for diversity δ_h , δ_r , and δ_e with respect to execution shift, execution pattern, and number of executions. Each box-and-whisker plot reports at the bottom the Wilcoxon test p-values between GA+CP and CP (first value), and between GA+CP and GA (second value).

very large subject systems with few tasks missing their deadlines, and very small systems where more deadline misses are revealed.

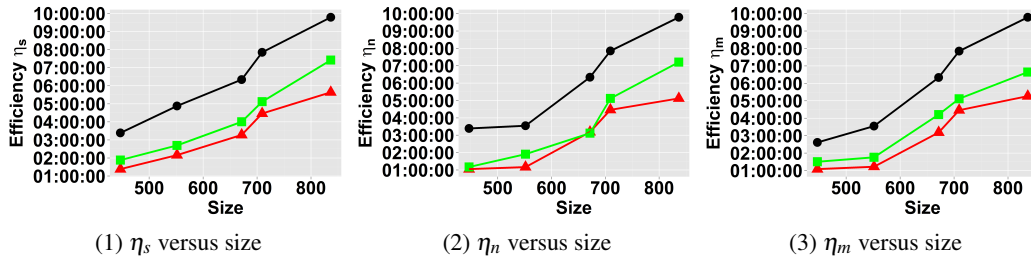


Fig. 8: Experimental results for efficiency η versus system size, comparing GA (▲), CP (●), and GA+CP (■)

6.5.5. Summary and Discussion. In light of these results, we conclude that, for the subject systems in our experiment, GA+CP has been nearly as efficient as GA and practically as effective as CP, while also generating solutions with a diversity similar to that of GA. Therefore, our results show that, within the range covered by our subject systems, GA+CP retains the advantages of both the efficiency and diversity of GA and the effectiveness of CP.

We conjecture that the reason for this result stems from the three factors discussed above. First, GA+CP is designed to perform a complete local search in the neighborhood of the best solutions computed by GA. Since GA+CP performs an additional search step over GA, it is expected that GA+CP requires more time than GA to find its best solutions. However, since the search performed during the CP step is confined in a neighborhood of restricted size, such local search is likely to terminate within a short time. Therefore, the time that CP spends improving the solutions found by GA is likely to have a negligible impact when compared to the time required by GA to find them. This allows GA+CP to achieve an efficiency which is only slightly worse than that of GA. Second, the search CP performs in the neighborhood of GA solutions is complete. Therefore, CP is certain to either find the best solution within distance D from the one GA computed, or to terminate proving that the solution found by GA is the best in its neighborhood. Furthermore, the search heuristics detailed in our previous work [Di Alesio et al. 2013] further improve the CP speed in optimizing the GA solutions. We performed a series of experiments on all subject systems varying the neighborhood size D , and empirically found out that a value of $D = 5$ was sufficient for GA+CP to achieve the same effectiveness as CP. However, we note that on different subject systems, CP might need to explore a larger neighborhood of GA solutions to reach the effectiveness of CP, and exploring such larger space might lead to a lower efficiency. Third, GA+CP is designed to search in the neighborhood of solutions computed by GA. This means that the local search performed by CP can find the same local optimum for different GA solutions (Figure 7.1), or more than one local optimum for a single GA solution (Figure 7.2). However, in our experiments these two scenarios did not have a significant impact on the diversity of solutions identified by GA, and resulted in GA+CP retaining the same diversity as GA.

6.6. Threats to Validity

We identified three main threats that could affect the general validity of our conclusions. First, the analysis of efficiency, effectiveness, diversity, and scalability is based on a set of five subject systems. Although evaluating GA+CP with respect to GA and CP in a larger number of systems would have mitigated this threat, the systems have been selected from different RTES domains and vary in size and complexity.

Second, the size of the subject systems selected varies from 6 to 32 tasks, 3 to 9 of which are aperiodic. There could be much larger systems featuring hundreds of tasks, and for those the efficiency

and effectiveness of GA+CP need to be investigated. This means that the conclusions drawn are valid only for systems in the same size range of the subject systems used in the comparison. To mitigate these first two threats, we could have manually constructed a set of systems with an increasing number of aperiodic tasks. In this way, we would have evaluated GA+CP in an arbitrarily large set of artificial subject systems, with the largest systems matching in size the most complex industrial RTES. However, this solution would have come at the cost of giving up the realistic nature of the five subject systems we considered.

Third, the experiment set-up relies on design choices that can potentially have a significant impact over the results. Specifically, we chose to run the search strategies for ten hours, and it is questionable whether a longer time could have led to significantly different results. However, by looking at the quartiles of the efficiency distributions, GA found its best results in significantly earlier than ten hours. This means that in most cases, GA reached a plateau before ten hours, and the chances for it to find a better solution if given more time are likely to be low. Furthermore, GA, and consequently GA+CP, rely on parameters specific to the domain of evolutionary algorithms, i.e., the initial population size, the crossover and mutation probabilities, and the population replacement rate. Values for these parameters different from the ones we used in the experiment could have led to significantly different results. However, we used the same values used by the strategy proposed by Briand et al. [Briand et al. 2006]. These values have been derived from the GA literature and specifically tuned for deadline miss analysis. Also note that, as opposed to GA, CP is fully deterministic, hence we expect the parameter sensitivity of GA+CP to be similar to that of GA. Finally, GA+CP also depends on the neighborhood size D where CP improves the solutions found by GA (Section 5.2). Our preliminary experimentation showed that a good compromise between efficiency and effectiveness is a value of D around 1% of T . To fully mitigate this threat, we would need a systematic investigation on the impact D has on efficiency, effectiveness, diversity and scalability.

7. CONCLUSIONS AND FUTURE WORK

Real-Time Embedded Systems (RTES) in safety critical domains have to react to external events within strict timing constraints. Failure to do so poses great risks for the system safety, as even a single task missing its deadline could result in a failure with severe consequences for the system itself, its users, and the environment. For this reason, systematic performance evaluation is of paramount importance to assess the system capability to operate safely. In RTES, the environment state plays a major role in determining the inputs and their timing, that for this reason can never be fully predicted prior to system execution. For assessing whether or not tasks will meet their deadlines at runtime, several approaches have been proposed. In particular, stress testing methodologies have been developed to identify scenarios that are likely to reveal deadline misses. Due to the large domain of system inputs, stress testing has often been cast as a search problem over the space of task arrival times. Both Genetic Algorithms (GA), and Constraint Programming (CP) have been used to solve this search problem, and have been shown to have their own practical advantages and drawbacks. Specifically, while GA is more efficient, i.e., faster in generating test cases, CP is more effective, i.e., it generates test cases that have a higher power to reveal deadline misses. Furthermore, GA also generates test cases that are more diverse, i.e., they have higher variety in terms of time span and preemptions between task executions, and numbers of aperiodic tasks executions.

To provide a strategy suitable for large and complex systems, in this paper we propose a combination of GA and CP aimed at retaining the advantages of the two search strategies. Specifically, we develop a combined GA+CP approach to support stress testing of task deadlines by identifying worst case scenarios where tasks are more likely to miss such deadlines. Similar to GA and CP in isolation, our approach expresses the generation of stress test cases as a search problem over the space of task arrival times. In this way, each solution to the problem, i.e., each sequence of arrival times for aperiodic task executions, characterizes one test case. The key idea behind our approach is to improve the solutions computed by GA by performing a complete search with CP in their neighborhood. Specifically, our approach consists of two separate stages. First, the search problem is solved through GA, using a state-of-the-art implementation for generating scenarios that are likely

to reveal task deadline misses [Briand et al. 2006]. This step produces an initial set of solutions, each characterizing a stress test case. Second, for each solution found by GA, CP searches in its neighborhood for better solutions through a Constraint Optimization Model derived from our previous work [Di Alesio et al. 2014]. This step produces the final set of solutions.

We evaluated GA+CP on five subject systems from different RTES domains with varying size and complexity. Our experimental validation highlighted four main results. First, GA+CP has an efficiency close to that of GA. This is because solutions are initially computed with GA, and the subsequent CP search is likely to terminate in a short time since it focuses on the neighborhood of a solution, rather than on the entire search space. Second, GA+CP has an effectiveness practically close to that of CP. This is because once GA has found a solution, CP further improves it by either finding a better solution, or by proving its optimality within the neighborhood. Third, GA+CP has a solutions diversity close to that of GA. This is because CP performs a local search in each neighborhood of a GA solution, and the GA solutions have high diversity. Fourth, we identified in the five systems of our experiment a linear trend between the efficiency, effectiveness, and diversity of GA+CP and the system size. Even though the scalability of GA+CP needs to be further ascertained, this result is encouraging, and is a significant step forward from our previous work [Di Alesio 2013] towards a stress testing approach suitable for industrial-size problems.

In the future, we plan to carry out the scalability analysis of GA+CP on even larger systems, and to continue exploring ways of combining complete and meta-heuristic search strategies to support stress testing. In this sense, we identified two main research directions. First, we plan to integrate in our approach a test suite optimization strategy to prioritize test cases that retain some property. For instance, starting from the test suite generated by GA+CP, we could find the minimal set of test cases that cover all the task executions predicted to miss a deadline. Such a strategy would allow us to investigate each potential deadline miss executing the least possible number of test cases. Second, we also plan to investigate multi-objective optimization to generate stress test cases that simultaneously exercise different performance properties of the system. Currently, our approach targets only task deadlines because we assume that even a single deadline miss poses a severe threat to the system safety. However, some real-time applications might be able to recover from short deadline misses, provided that enough computational resources are available. For these systems, we could for instance modify the fitness/objective function of GA+CP in order to generate test cases that push tasks to miss their deadlines while at the same time leading to high CPU usage. Such test cases would be able to uncover scenarios where deadline misses are more severe, because they happen when the system has insufficient CPU resources to recover in time.

Acknowledgments. The first and last authors acknowledge the Research Council of Norway (NFR 205606 - ModelFusion Project). The second and third authors are supported by the National Research Fund, Luxembourg (FNR/P10/03 - Verification and Validation Laboratory).

REFERENCES

- Wasif Afzal, Richard Torkar, and Robert Feldt. 2009. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology* 51, 6 (2009), 957–976.
- Rajeev Alur, Costas Courcoubetis, and David Dill. 1990. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on*. IEEE, 414–425.
- Saoussen Anssi, Sara Tucci-Piergiovanni, Stefan Kuntz, Sébastien Gérard, and François Terrier. 2011. Enabling scheduling analysis for AUTOSAR systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing, 2011 14th IEEE International Symposium on*. IEEE, 152–159.
- Krzysztof Apt. 2003. *Principles of constraint programming*. Cambridge University Press.
- Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering, 33rd International Conference on*. IEEE, 1–10.
- Alper Atamtürk and Martin WP Savelsbergh. 2005. Integer-programming software systems. *Annals of Operations Research* 140, 1 (2005), 67–124.
- Neil C Audsley, Alan Burns, Mike F Richardson, and Andy J Wellings. 1991. Real-time scheduling: the deadline-monotonic approach. In *Proc. IEEE Workshop on Real-Time Operating Systems and Software*. Citeseer.

- Theodore P Baker. 2006. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Systems* 32, 1-2 (2006), 49–71.
- Philippe Baptiste, Claude Le Pape, and Wim Nuijten. 2001. *Constraint-based scheduling: applying constraint programming to scheduling problems*. Vol. 39. Springer.
- Cornel Barna, Marin Litoiu, and Hamoun Ghanbari. 2011. Model-based performance testing: NIER track. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 872–875.
- Mohamad Bayan and João W Cangussu. 2008. Automatic feedback, control-based, stress and load testing. In *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 661–666.
- B. Beizer. 2002. *Software testing techniques*. Dreamtech Press.
- Donald J Berndt and Alison Watkins. 2005. High volume software testing using genetic algorithms. In *System Sciences, 2005. Proceedings of the 38th Annual Hawaiï International Conference on*. IEEE, 318b–318b.
- Lionel C Briand, Yvan Labiche, and Marwa Shousha. 2006. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines* 7, 2 (2006), 145–170.
- Simon Brown. 2000. Overview of IEC 61508. Design of electrical/electronic/programmable electronic safety-related systems. *Computing & Control Engineering Journal* 11, 1 (2000), 6–12.
- Hadrien Cambazard, Pierre-Emmanuel Hladik, Anne-Marie Déplanche, Narendra Jussien, and Yvon Trinquet. 2004. Decomposition and learning for a hard real time task allocation problem. In *Principles and Practice of Constraint Programming-CP 2004*. Springer, 153–167.
- Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. 2012. Model checking and the state explosion problem. In *Tools for Practical Software Verification*. Springer, 1–30.
- Alexandre David, Jacob Illum, K Larsen, and Arne Skou. 2010. Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1. *Model-Based Design for Embedded Systems* (2010), 93.
- Concettina Del Grosso, Giuliano Antoniol, Massimiliano Di Penta, Philippe Galinier, and Ettore Merlo. 2005. Improving network applications security: a new heuristic to generate stress testing data. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*. ACM, 1037–1043.
- Stefano Di Alesio. 2013. *The deadline misses constraints in ILOG solver v2*. Technical Report. <http://home.simula.no/~stefanod/ilog2.pdf>
- Stefano Di Alesio, Arnaud Gotlieb, Shiva Nejati, and Lionel Briand. 2012. 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, 764–769.
- Stefano Di Alesio, Shiva Nejati, Lionel Briand, and Arnaud Gotlieb. 2013. Stress testing of task deadlines: A constraint programming approach. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 158–167.
- Stefano Di Alesio, Shiva Nejati, Lionel Briand, and Arnaud Gotlieb. 2014. Worst-case Scheduling of Software Tasks – A Constraint Optimization Model to Support Performance Testing. In *Principles and Practice of Constraint Programming (CP 2014)*.
- Filippo Focacci, François Laburthe, and Andrea Lodi. 2003. Local search and constraint programming. In *Handbook of metaheuristics*. Springer, 369–403.
- Gordon Fraser, Andrea Arcuri, and Phil McMinn. 2014. A Memetic Algorithm for whole test suite generation. *Journal of Systems and Software* (2014).
- Vahid Garousi, Lionel C Briand, and Yvan Labiche. 2008. Traffic-aware stress testing of distributed real-time systems based on UML models using genetic algorithms. *Journal of Systems and Software* 81, 2 (2008), 161–185.
- Hassan Gomaa. 2006. Designing concurrent, distributed, and real-time applications with UML. In *Proceedings of the 28th International Conference on Software Engineering*. ACM, 1059–1060.
- Daniel Guimarans, Rosa Herrero, Daniel Riera, Angel A Juan, and Juan José Ramos. 2011. Combining probabilistic algorithms, constraint programming and lagrangian relaxation to solve the vehicle routing problem. *Annals of Mathematics and Artificial Intelligence* 62, 3-4 (2011), 299–315.
- Mark Harman and Phil McMinn. 2010. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Software Engineering, IEEE Transactions on* 36, 2 (2010), 226–247.
- Pascal Van Hentenryck and Laurent Michel. 2009. *Constraint-based local search*. The MIT Press.
- Thomas A Henzinger. 2013. Quantitative reactive modeling and verification. *Computer Science-Research and Development* 28, 4 (2013), 331–344.
- Pierre-Emmanuel Hladik, Hadrien Cambazard, Anne-Marie Déplanche, and Narendra Jussien. 2008. Solving a real-time allocation problem with constraint programming. *Journal of Systems and Software* 81, 1 (2008), 132–149.
- Abdollah Homaifar, Charlene X Qi, and Steven H Lai. 1994. Constrained optimization via genetic algorithms. *Simulation* 62, 4 (1994), 242–253.

- Kakuzo Iwamura and Baoding Liu. 1996. A genetic algorithm for chance constrained programming. *Journal of Information and Optimization sciences* 17, 2 (1996), 409–422.
- R. Jain. 2008. *The art of computer systems performance analysis*. John Wiley & Sons.
- Hermann Kopetz. 2011. *Real-time systems: design principles for distributed embedded applications*. Springer.
- Philippe Laborie. 2009. 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*. Springer, 148–162.
- Kiran Lakhota, Phil McMin, and Mark Harman. 2010. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of Systems and Software* 83, 12 (2010), 2379–2391.
- Claude Le Pape and Philippe Baptiste. 1997. An experimental comparison of constraint-based algorithms for the preemptive job shop scheduling problem. In *CP97 Workshop on Industrial Constraint-Directed Scheduling*. Citeseer.
- Fan Liu, Ajit Narayanan, and Quan Bai. 2000. *Real-time systems*. Citeseer.
- C. D. Locke, D. R. Vogel, L. Lucas, and J. B. Goodenough. 1990. *Generic avionics software specification*. Technical Report. DTIC Document.
- Arnaud Malapert, Hadrien Cambazard, Christelle Guéret, Narendra Jussien, André Langevin, and Louis-Martin Rousseau. 2012. An optimal constraint programming approach to the open-shop problem. *INFORMS Journal on Computing* 24, 2 (2012), 228–244.
- M. Mikučionis, K. G. Larsen, J. I. Rasmussen, B. Nielsen, A. Skou, S. U. Palm, J. S. Pedersen, and P. Hougaard. 2010. Schedulability analysis using UPPAAL: Herschel-Planck case study. In *Leveraging Applications of Formal Methods, Verification, and Validation*. Springer, 175–190.
- Nenad Mladenović and Pierre Hansen. 1997. Variable neighborhood search. *Computers & Operations Research* 24, 11 (1997), 1097–1100.
- Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.
- Shiva Nejati, Stefano Di Alesio, Mehrdad Sabetzadeh, and Lionel Briand. 2012. Modeling and analysis of cpu usage in safety-critical embedded systems to support stress testing. In *Model Driven Engineering Languages and Systems*. Springer, 759–775.
- Robert Nilsson, Jeff Offutt, and Jonas Mellin. 2006. Test case generation for mutation-based testing of timeliness. *Electronic Notes in Theoretical Computer Science* 164, 4 (2006), 97–114.
- Marie-Agnès Peraldi-Frati and Yves Sorel. 2008. From high-level modelling of time in MARTE to real-time scheduling analysis. *ACESMB* (2008), 129.
- Gilles Pesant and Michel Gendreau. 1996. A view of local search in constraint programming. In *Principles and Practice of Constraint Programming—CP96*. Springer, 353–366.
- Günther R Raidl. 2006. A unified view on hybrid metaheuristics. In *Hybrid Metaheuristics*. Springer, 1–12.
- M. Shams, D. Krishnamurthy, and B. Far. 2006. A model-based approach for testing the performance of web applications. In *Proceedings of the 3rd International Workshop on Software Quality Assurance*. ACM, 54–61.
- Paul Shaw. 1998. Using constraint programming and local search methods to solve vehicle routing problems. In *Principles and Practice of Constraint Programming—CP98*. Springer, 417–431.
- Abhishek Singh. 2009. *Identifying Malicious Code Through Reverse Engineering*. Vol. 44. Springer Science & Business Media.
- Brinkley Sprunt, Lui Sha, and John Lehoczky. 1989. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems* 1, 1 (1989), 27–60.
- Ken Tindell and John Clark. 1994. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and microprogramming* 40, 2 (1994), 117–134.
- Karim Traore, Emmanuel Grolleau, and Francis Cottet. 2006. Simpler analysis of serial transactions using reverse transactions. In *Autonomic and Autonomous Systems, International Conference on*. IEEE, 11–11.
- Elaine J Weyuker and Filippos I Vokolos. 2000. Experience with performance testing of software systems: issues, an approach, and case study. *Software Engineering, IEEE Transactions on* 26, 12 (2000), 1147–1156.
- Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, and others. 2008. The worst-case execution-time problem—An overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 36.
- Young-Su Yun and Mitsuo Gen. 2002. Advanced scheduling problem using constraint programming techniques in SCM environment. *Computers & Industrial Engineering* 43, 1 (2002), 213–229.
- Jian Zhang and SC Cheung. 2002. Automated test case generation for the stress testing of multimedia systems. *Software: Practice and Experience* 32, 15 (2002), 1411–1435.

Received Received; revised Revised; accepted Accepted