Genet Program Evolvable Mach (2006) 7:145–170
DOI 10.1007/s10710-006-9003-9

Using genetic algorithms for early schedulability analysis and stress testing in real-time systems

Lionel C. Briand · Yvan Labiche · Marwa Shousha

Received: 5 October 2005 / Revised: 7 December 2005 / Published online: 24 June 2006

Abstract Reactive real-time systems have to react to external events within time constraints: Triggered tasks must execute within deadlines. It is therefore important for the designers of such systems to analyze the schedulability of tasks during the design process, as well as to test the system's response time to events in an effective manner once it is implemented. This article explores the use of genetic algorithms to provide automated support for both tasks. Our main objective is then to automate, based on the system task architecture, the derivation of test cases that maximize the chances of critical deadline misses within the system; we refer to this testing activity as stress testing. A second objective is to enable an early but realistic analysis of tasks' schedulability at design time. We have developed a specific solution based on genetic algorithms and implemented it in a tool. Case studies were run and results show that the tool (1) is effective at identifying test cases that will likely stress the system to such an extent that some tasks may miss deadlines, (2) can identify situations that were deemed to be schedulable based on standard schedulability analysis but that, nevertheless, exhibit deadline misses.

Keywords Software verification and validation · Schedulability theory · Genetic algorithms

L. C. Briand (✉) · Y. Labiche · M. Shousha
Software Quality Engineering Laboratory, Department of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada
e-mail: briand@sce.carleton.ca

Y. Labiche
e-mail: labiche@sce.carleton.ca

M. Shousha
e-mail: mshousha@connect.carleton.ca

1. Introduction

An increasing number of software applications are concurrent in nature. This often entails that activities/tasks occur/execute in parallel and the order of the incoming events triggering those activities/tasks is often not predictable [5]. This is particularly true for real-time and distributed systems.

Specific methods have been proposed for the development of real-time systems. They often suggest that, during development, some performance analysis be performed to determine whether designed tasks will likely meet their deadlines [5]. At design time, such performance analysis requires that task execution times be estimated (e.g., based on the expected number of lines of code), since the whole software is likely not fully developed. Real-Time Scheduling Theory [5, 16] helps designers determine whether a group of tasks (periodic or aperiodic, possibly with synchronizations and priorities), whose individual execution times have been estimated, will meet their deadlines. However, when dealing with aperiodic tasks, the proposed techniques make an important assumption: Aperiodic tasks are transformed, for the purpose of schedulability analysis, into periodic tasks whose periods are equal to the minimum inter-arrival times of the events that activate the aperiodic tasks [19].

Because of inaccuracies in execution time estimates and simplifying assumptions regarding aperiodic tasks, we cannot simply rely on schedulability theory alone. It is then important, once a set of tasks have been shown to be schedulable, to derive test cases to exercise the system and verify that tasks cannot miss their deadlines, even under the worst possible circumstances. Our first objective is thus to exercise the system in such a way that some tasks are close to missing a deadline. We refer to this testing activity as *stress testing*, a definition consistent with that provided in [1]: "subjecting the system to harsh inputs [. . .] with the intention of breaking it".

The stress testing strategy presented in this paper aims at finding combinations of inputs such that completion times of a specific task's executions are as close as possible to their deadlines. These input combinations, i.e., test cases, should account for both seeding times and possible input data for aperiodic tasks, since both impact task executions (e.g., varying input data may trigger different control flow and result in varying execution times). We, however, limit our study to seeding times for aperiodic tasks, since input data are usually accounted for in execution time estimates [9]. Finding such test cases is not easy as many periodic and aperiodic tasks, with different priorities, can be triggered. But, if a practical way to automate test cases is found, it would allow us to specify a scenario of event arrival times that makes it more likely for tasks to miss their deadlines if their execution time estimates turn out to be too inaccurate, once the system is implemented.

The search for an optimal combination of inputs uses a Genetic Algorithm (GA), as the problem to be solved is an optimization problem (the execution end of a task must be as close as possible to the deadline) under constraints (e.g., priorities). The specification of test cases for stress testing does not require any (running) implementation to be available, and can thus be derived during design, once a task architecture is defined (e.g., specifying estimated execution times, priorities). This can occur just after schedulability analysis, once tasks have been deemed schedulable using Real-Time Scheduling Theory. Stress testing can then be planned early and, once the implementation is available and after the completion of functional testing, stress testing may begin right away.

This paper also proposes another application of our GA-based task analysis. Given the assumption on which Real-Time Scheduling Theory relies for the analysis of aperiodic tasks, we show how to use our GA-based approach to relax this assumption and further investigate whether tasks are schedulable in a more realistic manner.

The rest of the article is structured as follows. Section 2 provides some background on schedulability analysis as well as some related work. Section 3 describes why we chose GAs and how we tailored them to address the objectives stated above. It also introduces a prototype tool implementing our strategy. Two case studies are then reported in Section 4 and we draw conclusions in Section 5.

2. Background and related work

2.1. Task scheduling strategies and schedulability theory

On a single processor, or CPU, concurrent tasks must be handled by the kernel of the operating system. The kernel maintains a list of tasks that are ready to execute based on different task scheduling algorithms. The round robin algorithm, for example, uses a FIFO list of tasks, while other algorithms use task priorities and preemption [5, 19]. The Portable Operating System Interface Standard (POSIX), to which most commercial real-time operating systems conform, assumes fixed preemptive priority scheduling.

Real-time scheduling theory is commonly used to determine whether a group of tasks will meet their deadlines. Only the tasks that are part of the interface of the software system under test (i.e. those triggered by events from users, other software systems or sensors) are involved in the analysis.

A task is schedulable if it always completes its execution before its deadline elapses. A group of tasks is schedulable if each task always meets all its deadlines. In the case of independent (no task communication or synchronization) periodic tasks, theorems determine whether tasks scheduled with the rate monotonic algorithm will always meet their deadlines. In the case of aperiodic or dependent tasks, the rate monotonic algorithm has to be adapted: (1) Aperiodic tasks must be modeled appropriately, and (2) Task priorities have to be adapted to avoid the blocking of high priority tasks by lower priority tasks, referred to as rate monotonic priority inversion. The Generalized Completion Time Theorem (GCTT) assumes fixed preemptive priority scheduling, as specified in the POSIX, which we will use for our case studies. GCTT extends the basic rate monotonic theory and can be used to determine whether a task can complete execution by the end of its period, given preemption by higher priority tasks and blocking time by lower priority tasks. For schedulability analysis, an extension exists for the GCTT that models aperiodic tasks as periodic ones. In this model, aperiodic tasks are ready to execute when the system starts executing and are triggered at regular time intervals as determined by their minimum inter-arrival times [5, 16, 19]. We do not further discuss these techniques and refer the interested reader to [5, 16, 19].

During the software design phase, real-time scheduling theory requires that task CPU utilization times, referred to as execution times in this article, be estimated, since the whole software is likely not fully implemented.[1] These execution time estimates for tasks that are part of the interface of the software system must account for tasks triggered by internal events (i.e., events triggered by external events and hidden to the outside of the software system). This is not an easy task as execution times depend, among other things, on the triggered control flow and on the underlying system (e.g., cache memory) [9]. At design time, analytic benchmarking and code profiling are the most common strategies employed to determine those estimates [8]. Analytic benchmarking measures the performance of selected hardware when representative code samples (primitive code types) are executed. This information is

[1] Note that task periods also have to be estimated.

then combined with code profiling—which decomposes a task into the same set of primitive code types—to determine the makeup of the task and estimate its execution time. (A similar strategy is suggested in [5].) As a result, for any given real-time system, the accuracy of estimates can vary widely, especially in the early design stages. Other strategies exist but require that tasks be implemented (e.g., statistical prediction [8]). Further discussions on strategies to estimate task execution times at design time are out of the scope of this article.

2.2. Related work

A number of papers have used GAs to generate test data, mostly in the context of structural testing (e.g., [20]). To the best of our knowledge, only one work addresses a goal similar to ours: To analyze real-time task architectures and determine whether deadlines can be missed [13]. The approach attempts to verify worst-case and best-case execution times of tasks in real-time systems, which can then be used for performance analysis. The approach combines testing, i.e., system execution using input values, and evolutionary computation (in this case GAs) and approximates these extreme execution times in a step-wise manner by executing the system of interest with varying input values. At each step, i.e., for each set of input values, a measure of the execution of the system (using these input values) is used to evaluate whether the algorithm gets closer to the optimum execution time. It is thus a technique to empirically verify, one task at a time, the execution time hypotheses made at design time for schedulability analysis. The approach, however, is different from ours as (1) it requires an implementation of the system under study and (2) it considers tasks in isolation (i.e., separately from the other tasks) and only focuses on violated timing constraints due to input values (e.g., the two matrices of a function that multiplies matrices). In this paper, we analyze task architectures and consider seeding times of events triggering tasks and tasks' synchronization. Both works are complementary as inputs will influence task execution times and task completion times will depend on triggering events' seeding times and tasks' synchronization.

Another related work [22] describes a procedure for automating test case generation for multimedia systems. The flow and concurrency control between nodes on the network is modeled using Petri nets (PN) coupled with temporal constraints. Test cases are then marking sequences in the PN model. Using linear programming, they are determined to potentially lead to resource saturation. The aim is twofold: To detect load sensitive faults and to ensure that systems adhere to their performance requirements under heavy loads. Our context is different as our target is reactive, concurrent, real-time systems with soft or hard deadlines. Our focus is on missing deadlines as opposed to resource saturation (even though resource saturation may lead to deadline misses, as explained in [22]).

3. Tailoring genetic algorithms

In order to achieve our objectives, we need to derive a sequence of seeding times for aperiodic tasks so that the difference between the end of execution and the deadline of a target task (selected by the test engineer[2]) is as close as possible to zero. This sequence of seeding times is to be chosen such that the delay between two consecutive seeding times of an

[2] This target task can be the most 'critical' task in the system, for instance (e.g., a missed deadline for the target task would have catastrophic results). This analysis can be repeated for all 'critical' tasks, leading to several test scenarios, each focusing on stress testing a specific task.

aperiodic task are greater than its minimum inter-arrival time and smaller than its maximum inter-arrival time (when it exists). When there are more than a few tasks, it then becomes clear that deriving such stress test cases is not trivial and needs to be automated.

A variety of methods exist for solving optimization problems. Among them is linear programming, which can be used when linear functions constrain the problem. Since, as we will see in Section 3.4, the optimization problem at hand can not be completely expressed using linear functions, linear programming does not seem to be an adequate choice. Other possible techniques are Simulated Annealing (SA) and Genetic Algorithms (GA) [7]. Many researchers seem to agree that because GAs maintain a population of possible solutions, they have a better chance of locating the global optimum compared to SA, which proceeds one solution at a time [11]. Hence, we adopt GAs as our optimization solution methodology.

The rest of the section first introduces our notation. Next, we describe how we tailor GAs for the problem at hand (Sections 3.2 to 3.5): encoding for chromosomes, cross-over and mutation operators, objective function, and the way we set the GA's parameters. Last, we describe how we implement our solution (Section 3.6).

3.1. Notation

All tasks, whether aperiodic (A_i) or periodic (P_i) have CPU execution estimates C_i. They also have priorities p_i. $A_{i,\mathrm{j}}$ and $P_{i,j}$ denote the jth execution of task i for aperiodic and periodic tasks, respectively, during the testing interval T. Each task execution j of task i has a deadline ($d_{i,j}$) at which it must complete its execution, an (external event) arrival time ($a_{i,j}$) at which the task can start running, an execution start time ($s_{i,j}$) at which the task actually starts running, and an execution end time ($e_{i,j}$) which determines the time unit at which it completes. The values of $a_{i,j}$ and $s_{i,j}$ do not always coincide. In fact, they only coincide if the corresponding task is the highest priority available task ready to execute.

Periodic tasks additionally have periods (r_i), and $d_{i,j}$ and $a_{i,j}$ are both multiples of those periods: $a_{i,j} = (j-1)r_i$ and $d_{i,j} = jr_i$. Aperiodic tasks, on the other hand, have minimum inter-arrival times min_i and possibly maximum inter-arrival times max_i indicating the minimum and maximum time intervals between two consecutive arrivals of the event triggering the task, respectively. Like periodic tasks, aperiodic tasks define deadlines d_i, and $d_{i,j} = a_{i,j} + d_i$.

In the testing time interval T, each task has a maximum number of executions k_i that depends on the period (periodic task) or minimum inter-arrival time (aperiodic task).

3.2. Chromosomes

The encoding of solutions to the problem at hand into a chromosome is paramount when specifying a GA, as this drives the ease to define (and eventually implement) mutation and crossover operators.

3.2.1. Coding

In our problem, the values to be optimized (i.e., the genes) are the arrival times of all aperiodic tasks. A gene can be depicted as a pair of integer values ($A_i, a_{i,j}$); that is an aperiodic task number and an arrival time. As the chromosome holds the arrival times of all aperiodic tasks, the chromosome's size (i.e., the number of genes) is the total number of executions of all aperiodic tasks. Since the maximum number of executions for aperiodic task i over a period

T is the ceiling of $T/\min_i$ ($k_i = \lceil T/\min_i \rceil$), the length of the chromosome is:

$$l = \sum_{i=1}^{n} \left\lceil \frac{T}{\min_i} \right\rceil,$$

where n is the number of aperiodic tasks.

Because constant chromosome sizes during the execution of a GA are considered good design (this facilitates the definition of operators), and because we may not need all k_i arrival times for task i, we use a special value for arrival times to depict a non-existent arrival time: -1.

The genes of the chromosome are subject to constraints, as two consecutive arrival times for a particular event must have a difference of at least the minimum inter-arrival time, and at most the maximum inter-arrival time (if it exists). If no maximum inter-arrival time is defined for an aperiodic task, it is set to T. Also, in order to facilitate chromosome manipulations, all genes corresponding to the same task are grouped together and ordered increasingly according to $a_{i,j}$. For example, given a set of two aperiodic tasks $t1$ ($\min_{t1} = 10$) and $t2$ ($\min_{t2} = 11$), the following is a valid chromosome in a time interval $T = 30$: ($t1, -1$) ($t1$, 19) ($t1$, 29) ($t2, -1$) ($t2, -1$) ($t2$, 10). However, chromosome ($t1$, 0) ($t1$, 5) ($t1$, 50) ($t2, -1$) ($t2, -1$) ($t2$, 10) is not valid since the minimum inter-arrival time constraint for the first task is not satisfied by the two first genes, and the last arrival time for the first task is above the testing time interval T.

3.2.2. Initialization

The initial population of chromosomes is randomly created, following the constraints above, provided that a value for T is given. The length of the chromosome and the number of arrival times for each task are computed from T (see formula above).

The value of $a_{i,j}$ is randomly selected from a range determined by the arrival time of $a_{i,j-1}$ (i.e., the previous arrival time for the task) as well as $\min_i$ and $\max_i$ (if $\max_i$ is not specified, its value is set to T): $[a_{i,j-1} + \min_i, a_{i,j-1} + \max_i]$. If there is no previous gene (i.e., $j = 1$), or the previous gene depicts a non-existent arrival time (i.e., $a_{i,j-1} = -1$), the range is $[0, \max_i]$. If the number selected from the range is greater than T, $a_{i,j}$ is considered non-existent and is set to -1, and the gene is moved before the first gene for the task (genes are ordered).

For example, consider the initialization for an aperiodic task $t1$ ($\min_{t1} = 10$) with $T = 30$: $k_{t1} = 3$. Three empty genes are created (1, ...) (1, ...) (1, ...). Because there is no previous gene, the value of $a_{1,1}$ is randomly chosen from the range $[0, \max_{t1}]$ (because $\max_{t1}$ is not specified, the range is $[0, T = 30]$. Assume this yields: (1,15) (1, ...) (1, ...). Similarly, for the second gene, the value of $a_{1,2}$ is randomly chosen from the range [25, 45] ($[a_{1,1} + \min_{t1}, a_{1,1} + \max_{t1}]$). Further assume the genes are now: (1,15) (1,27) (1,...). Initialization proceeds similarly for the third gene with the value of $a_{1,3}$ chosen from [37, 57]. Any value in this range is greater than the value of T (30). The value of the third gene is thus set to -1 and the gene is moved to the beginning of the chromosome yielding (1, -1) (1,15) (1,27).

3.3. Operators

Cross-over and mutation operators are the ways GAs explore a solution space [7]. Hence, they must be formulated in such a way that they efficiently and exhaustively explore the

solution space. If the application of an operator yields no change or an invalid chromosome, backtracking is performed to first invalidate the operation, then to reapply the operator. Backtracking, however, is deemed expensive and can seriously slow down executions of the GA. Moreover, some GA tools incorporate backtracking while others do not. To allow for generality, we assume no backtracking methodology is available. Hence, in our implementation of the operators, we formulate our own backtracking methodology. More precisely, if the application of the operator does not alter the chromosome, we do not commit the changes and search for a different chromosome or gene within the chromosome—depending on the operator—and reapply the operator.

Formulating operators is rather a difficult task, as genetic operators must maintain allowability. In other words, genetic operators must be designed in such a way that if a constraint is not violated by the parents, it will not be violated by the children resulting from the operations [4].

3.3.1. Cross-over operator

Crossover operators aim at passing on desirable traits or genes from generation to generation [7]. Varieties of crossover operators exist, such as sexual, asexual and multiparent [21]. The former uses two parents to pass traits to the two resulting children. Asexual crossover involves only one parent and produces one child that is a replica of the parent. Any differences between parent and offspring are the result of errors in copying or mutations. These, however, occur very rarely. Multiparent crossover, as the name implies, combines the genetic makeup of three or more parents when producing offspring. Different GA applications call for different types of crossover operators. We employ the most common of these operators: sexual crossover. The general idea behind sexual crossover, also called n-point crossover [14], is to divide both parent chromosomes into two or more fragments and create two new children by mixing the fragments [7]. In n-point crossover, the two parent chromosomes are aligned and cut into $n + 1$ fragments at the same places. Once the division points are identified in the parents, two new children are created by alternating the genes of the parents [14].

To avoid mixing tasks executions and violating constraints too often (e.g., mixing genes for different tasks), our cross-over operator is an n-point cross-over [14] where n is the number of aperiodic tasks being scheduled. The actual division points of the parents depend on k_i for each task i: The first point of division occurs after k_{t1}; the second point of division after $k_{t1} + k_{t2}$ from the first gene; the $n-1$ point of division after $k_{t1} + k_{t2} + \cdots + k_{tn-1}$ from the first gene.

Once the division points are identified in the parents, two new children are created by inheriting fragments from parents with a 50% probability. In other words, for each pair of fragments $f1$ and $f2$ of the same task belonging respectively to parents 1 and 2, child 1 inherits $f1$ with a 50% probability, and if it does inherit $f1$, child 2 inherits $f2$, and conversely.

Let us consider an example with three aperiodic tasks $t1$ ($\min_{t1} = 100$), $t2$ ($\min_{t2} = 150$) and $t3$ ($\min_{t3} = 200$). The operator is a two-point crossover as there are three tasks. Table 1 shows two parent chromosomes and the generated offspring by the two-point crossover. The shaded areas indicate which genes in child 1 come from which parent. Child 1 inherits the first and third fragments from parent 1, and the second fragment from parent 2.

3.3.2. Mutation operator

Mutation aims at altering the population to ensure that the genetic algorithm avoids being caught in local optima. The process of mutation proceeds as follows: a gene is randomly

Table 1 Crossover operator—an example

	Task $t1$	Task $t2$	Task $t3$
Parent 1	**($t1$, 25) ($t1$, 150)**	($t2$, −1) ($t2$, 150)	**($t3$, 0)**
Parent 2	($t1$, 5) ($t1$, 200)	**($t2$, 50) ($t2$, 200)**	($t3$, 55)
Child 1	**($t1$, 25) ($t1$, 150)**	**($t2$, 50) ($t2$, 200)**	**($t3$, 0)**
Child 2	($t1$, 5) ($t1$, 200)	($t2$, −1) ($t2$, 150)	($t3$, 55)

chosen for mutation; the gene is mutated, and the resulting chromosome is evaluated for its new fitness. We define a mutation operator that mutates genes in a chromosome by altering their arrival times. The idea behind this operator is to move task executions within the time interval $[0, T]$, i.e., closer to the next or previous task execution so as to increase the likelihood of missed deadlines. Like the cross-over operator, this is done in such a way that the constraints on the chromosomes are met (see Section 3.2).

Effectively, gene j modeling an execution of a task i is randomly selected from a chromosome. A new arrival time $a'_{i,j}$ is chosen for it from the range $[a_{i,j-1} + \min_i, a_{i,j-1} + \max_i]$ (or $[0, \max_i]$ if the gene is the first in the segment or if the previous gene has arrival time -1). New values are also generated using $a'_{i,j}$ for subsequent executions of the same task that no longer uphold the inter-arrival time constraints. This is done in a manner similar to the initialization of the population. Indeed, when changing arrival time j for task i, arrival time $j + 1$ might fall outside the permissible range of $[a'_{i,j} + \min_i, a'_{i,j} + \max_i]$. If arrival time $a'_{i,j}$ is greater than T, then its value is set to -1 and the gene is moved to the beginning of all the genes for task i. Furthermore, if after mutation the difference between the last arrival time of task i and T is greater than $\max_i$, then there is room for (at least) one additional arrival time. A gene with seeding time -1 is modified to fill the gap. (Our construction of chromosomes and modifications with cross-over and mutation operators ensures such a gene with arrival time equal to -1 exists.) This last check ensures that the last execution of a task is a valid ending execution and that no other executions should occur after it.

In case the gene chosen for mutation within a chromosome has arrival time equal to -1, that gene is eliminated and is replaced by a new gene. The overall effect is the addition of a task execution. When inserting a new task execution, every two consecutive task executions are examined (starting from the first gene with arrival time different from -1) to determine whether an insertion between them will not violate either minimum or maximum inter-arrival times. If this is the case, the new gene is inserted in that location. Otherwise, the remaining consecutive task executions are examined. When examining the first execution of the task sequence, that is the first gene with arrival time different from -1, insertion can only occur before this gene, say gene x, if the value of the arrival time is greater than or equal to the minimum inter-arrival time of the task. This indicates that values lying in the range $[0, a_{i,x} - \min_i]$ may be inserted before j while still upholding minimum and maximum inter-arrival time constraints. Similarly, for consecutive genes x and $x + 1$, a new gene can be inserted in between if and only if $a_{i,x+1} - a_{i,x} \geq 2 * \min_i$. Values lying in the range $[a_{i,x} + \min_i, a_{i,x+1} - \min_i]$ may be inserted between x and $x + 1$ while still upholding the time constraints. When examining the last gene of the fragment, a new gene with values lying in the range $[a_{i,x} + \min_i, a_{i,x} + \max_i]$ can be inserted after it. Insertions thus occur from left to right along the executions of a task. If no suitable insertion location is found, this inherently means that no task execution can be added among the already existing task executions. A different gene is then randomly chosen for mutation.

Let us consider an example with three aperiodic tasks $t1$ ($\min_{t1} = 200$), $t2$ ($\min_{t2} = 150$, $\max_{t2} = 200$) and $t3$ ($\min_{t3} = 300$), and $T = 400$. A sample chromosome composed of six genes (numbered 1 to 6) is: ($t1$, -1) ($t1$, 200) ($t2$, 50) ($t2$, 200) ($t2$, 375) ($t3$, 0). Assuming that gene 4 is randomly chosen for mutation, a new value is chosen from the range [200, 250] ([50 + 150, 50 + 200]), e.g., 220. This value is less than the value of $T = 400$; hence it is acceptable. The chromosome is now: ($t1$, -1) ($t1$, 200) ($t2$, 50) ($t2$, 220) ($t2$, 375) ($t3$, 0). Because gene 4's value was altered, all subsequent genes may have to be modified if inter-arrival time constraints are not satisfied. Here, gene 4 and 5 arrival times satisfy the constraints, i.e., gene 5's arrival time (375) is greater than the sum of gene 4's arrival time (220) and the minimum inter-arrival time for the task (150). The mutation operation stops. A randomly selected value of 230 for gene 4's new arrival time leads to a different situation. The altered chromosome is now ($t1$, -1) ($t1$, 200) ($t2$, 50) ($t2$, 230) ($t2$, 375) ($t3$, 0). Gene 5's arrival time should then lie in range [230 + 150, 230 + 200], that is [380, 430], which is not the case. A new value for gene 5's arrival time is thus randomly selected from range [380, 430]. Assuming the selected value is 390, the chromosome becomes ($t1$, -1) ($t1$, 200) ($t2$, 50) ($t2$, 230) ($t2$, 390) ($t3$, 0) and the mutation stops. If the randomly selected value is instead 420, which is greater than $T = 400$, gene 5's arrival time is set to -1 and the gene is moved to the beginning of the genes for the task, resulting in chromosome ($t1$, -1) ($t1$, 200) ($t2$, -1) ($t2$, 50) ($t2$, 230) ($t3$, 0).

Reusing the same example, assume gene 1 is randomly selected for mutation. Because the arrival time of gene 1 is -1, we eliminate this gene and insert a new execution into the sequence of task executions. We examine the next gene, gene 2, with an arrival time value of 200. This value is equal to $\min_{t1}$. Thus, we can insert the new execution before it. The range of choice is [0, 0] ([0, 200 – 200]). The mutated chromosome thus becomes: ($t1$, 0) ($t1$, 200) ($t2$, 50) ($t2$, 200) ($t2$, 375) ($t3$, 0).

It is important to note that because this mutation operator allows task executions to move along the specified time interval T, in both directions, it effectively explores the solution space. Furthermore, backtracking is eliminated by not committing any changes unless the changes lead to a valid chromosome.

3.4. Objective function

Recall from Section 1 that our optimization problem is defined as follows: What sequence of arrival times for aperiodic tasks will cause the greatest delays in the executions of the target task (e.g., a selected periodic or aperiodic critical task)? In defining the delay in the target task's executions, we consider the difference between the deadline of an execution and the execution's actual completion, i.e., $d_{t,j} - e_{t,j}$ for target task t. We are thus interested in rewarding smaller values of the difference and penalizing larger values. Assuming scheduled tasks may miss deadlines, we also have to reward negative values over positive values. Note that in order to have executions' actual completion times (i.e., values for $e_{i,j}$), we need to implement a scheduler that schedules the tasks at hand given the arrival times described in a chromosome, and the task architecture (in particular, CPU time estimates).

Given these requirements, we considered a number of solutions [3]. The objective function we found that best suits our criteria is an exponential function:

$$f(ch) = \sum_{j=1}^{k_t} 2^{e_{t,j} - d_{t,j}},$$

where Ch is the chromosome and t is the target task.

Note that in most cases, task executions meet deadlines, thus resulting in negative values for $e_{t,j} - d_{t,j}$; that is small values for $f(Ch)$. Also, the greater the difference between deadline and end of execution, the smaller the value of $f(Ch)$. Moreover, missed deadlines result in positive values for $e_{t,j} - d_{t,j}$. In other words, larger values of $f(Ch)$ are indicative of fitter individuals. This fitness function thus has to be maximized by the GA. The objective function is expressed in exponential form to prevent the overshadowing of one bad execution with a large deadline miss by many good executions that meet their deadlines.

It is also important to note that the objective function is for one unique task, which is either periodic or aperiodic. This allows us to focus on the most time critical task in the system and ensure that its deadlines are all met. Recall from Section 3.1 that $d_{i,j} = jr_i$ and $d_{i,j} = a_{i,j} + d_i$ for periodic and aperiodic tasks, respectively. Determining $e_{t,j}$ requires that all the tasks (periodic and aperiodic) be scheduled, given the aperiodic tasks' arrival times specified in the chromosome.

3.5. Setting GA parameters

In addition to deciding on an encoding, mutation and cross-over operators and a fitness function, a number of parameters must be set for the GA to produce nearly optimal solutions to the problem.

First, the replacement strategy we use is steady state replacement [7], with which the population size does not change and a fixed number of chromosomes are changed each time the population evolves. The replacement percentage we apply is 50%, which complies with the findings reported in [7]. The selection strategy for choosing an individual for mutation and crossover is the roulette wheel selection method, in which fitter chromosomes are more likely to be selected to produce offspring [7].

Throughout the GA literature, various mutation rates, crossover rates and population sizes have been used [7]. Of the common mutation rates, those that take the length of the chromosome and population size into consideration perform significantly better than those that do not [17]. We thus apply the mutation rate suggested in [7], i.e., $\frac{1.75}{\lambda\sqrt{l}}$, where λ denotes the population size and l is the length of the chromosome [15]. Similarly, consistent with the observations reported in [7], we apply a crossover rate of 70%, and the population size is 80.

Two kinds of termination criteria have traditionally been used for GAs and all require the setting of parameters [12]: (1) A maximum number (to be determined) of generations or evaluations of the fitness function is reached; (2) The chance for a significant improvement is relatively small, e.g., a plateau seems to be reached, or there is low population diversity. The former is simpler to implement but the latter is adaptive as it monitors population characteristics (e.g., the variety of genes in the population) across generations. Since our work is at the proof of concept stage, we decided to use a number of generations as a termination criterion. Alternatively, adaptive criteria will be considered in future work. Once 500 generations have been generated, the GA halts, yielding the best score found. This number is based on experimentation: We ran a number of tests on the application with various values for the stopping criterion and found the best value to be 500.

3.6. Prototype tool

Following the principles described in the previous sections, we have built a prototype tool called Real-Time Test Tool (RTTT). Three inputs must be provided by the user: (1) for each

```
Once RTTT has identified a set of seeding times S for aperiodic
tasks using target task t:
1.    Increase each (a)periodic task execution by 0.1%
2.    Schedule the tasks using S and new (increased) execution
      times
3.    Repeat steps 1 and 2 until target task t misses a deadline
4.    Report the total increased percentage
```

Fig. 1 Algorithm to determine the maximum inaccuracy percentage for the target task

task, the task information, comprised of a task number, priority, estimated execution time, dependencies to other tasks in the form of shared resource dependencies [5] (if any), a period for a periodic task or minimum and (possibly) maximum inter-arrival times and deadline for an aperiodic task; (2) test environment information comprised of the time interval during which the test is to be performed as well as the target task; and (3) whether the tool should output a timing diagram corresponding to the result. (A timing diagram is a diagram that shows the time-ordered execution sequence of a group of tasks—see Section 4 for examples.)

It is worth noting that the tool can be used whether or not the group of tasks under test, both periodic and aperiodic, are schedulable under the Generalized Completion Time Theorem and its extension. If they are deemed schedulable, RTTT will attempt to confirm whether this is really the case. If, on the other hand, they cannot be deemed schedulable by the GCTT and its extension, i.e., we do not know whether the tasks are schedulable, we use RTTT to investigate whether we can find a sequence of arrival times where deadline misses occur. If we cannot find such a sequence, this does not guarantee that none exist. However, one can still feel more confident that such a case is unlikely.

The tool reports two or three different results, depending on the third input. The first result is the sequence of arrival times for all aperiodic tasks (i.e., the chromosome), with the best fitness function value obtained by the GA. The second result is a measure of *safe estimate percentage* for the target task, as obtained by using the algorithm in Fig. 1. This indicates the maximum inaccuracy percentage that can be made during the estimation of all tasks' execution time without deadline misses occurring for the target task. The user can then assess whether such inaccuracies are realistic or unlikely, thus assessing the risk of encountering response time problems at run time. Further actions may include additional performance studies to estimate certain tasks' execution times more precisely. Note that RTTT assumes inaccuracies to be the same for all tasks. Though this may not be the case in practice, this is a necessary simplifying assumption to ease the construction of our algorithms. Additionally, it is difficult to foresee a generic mechanism to predict accuracy differences for tasks. Finally, the third result consists of two timing diagrams: one is the GA solution maximizing the fitness function, and the other is a scheduling of the tasks while accounting for the safe estimate percentage determined by RTTT.

The RTTT prototype executes using two modules. The first one is a GA. Its implementation is an instance of GAlib, a framework written is C ++ for the creation of GAs [21]. The second module is a POSIX [5] compliant scheduler emulating single processor execution. This is required, as the fitness function used in the GA requires end times of task executions. In other words, each time the fitness function is evaluated for a chromosome, the aperiodic tasks in the chromosome, as well as the periodic tasks described in the inputs, are scheduled by the scheduler. The scheduler is a fixed preemptive priority scheduler, assuming single processor execution. Task dependencies in our application are in the form of shared resource

dependencies [5]. Hence, two tasks are dependent if they share a common resource. If a dependency occurs between tasks, the first ready task of the dependency must fully complete its execution before the dependent task can run, regardless of its priority. Equal priority tasks are executed in a first-come-first-served fashion. It is worth noting that the tool can easily be adapted to other scheduling strategies, as only this second module is then to be changed. The reader interested in more technical details in referred to [3].

4. Applications and case studies

This section discusses applications of the RTTT tool and presents three case studies. The first one (Section 4.1) is representative of the many scenarios we have used to test our tool, and illustrates how RTTT can be used to generate seeding times that bring the target completion times closer to their respective deadlines. We further show that a small error in the estimated execution times of the system tasks can then lead, at testing time, to missing deadlines. This suggests that RTTT can be used to generate test cases that will stress the system more than the scenarios entailed by schedulability theory (e.g., GCTT). Our second example shows that RTTT sometimes identifies seeding times that will lead to missed deadlines, even when tasks have been determined to be schedulable, and the execution time estimates are accurate (Section 4.2). In the third example (Section 4.3) we show the usefulness and feasibility of the approach on an actual real-time system whose task architecture is public domain [10], confirming the previous two results observed on artificial examples.

As GAs are a heuristic optimization technique, variance occurs in the results produced by different GA executions. To assess the extent of such variability, each case study was run 10 times and we studied the variance in both the objective function and the difference between execution end and deadline. Average execution times are also reported for each case study, running on an 800 MHz AMD Duron processor with 192 KB on-chip cache memory and 64 MB DRAM memory. GAs can be computationally expensive and their efficiency needs to be reported to demonstrate their practicality for a given problem.

4.1. Execution time estimates must be accurate

Let us consider the three tasks $t1$ (periodic), $t2$ (aperiodic) and $t3$ (periodic) whose characteristics are shown in Table 2: Task $t1$ has a higher priority than $t2$ and $t3$.

Using GCTT and its extension, we can prove that these three tasks are schedulable (details are not provided here). This can be illustrated with a timing diagram, i.e., a diagram that shows the time-ordered execution sequence of a group of tasks (see Fig. 2(a)). The timing diagram notation we use is an adaptation of the one proposed in [5]: Time appears on the left of the diagram, from top to bottom, events triggering tasks are shown on the right (with arrows), shaded rectangles indicate when tasks execute, and task preemption is shown

Table 2 Task characteristics: Example 1

	Task $t1$	Task $t2$	Task $t3$
Period (periodic), or minimum inter-arrival time (aperiodic)	255	240	250
Priority	32	31	30
Execution time	200	20	20

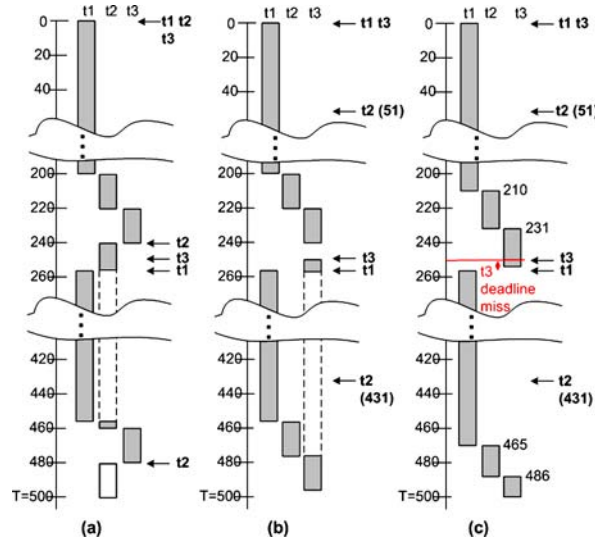

Fig. 2 Timing diagrams for Example 1: Execution times must be accurate

with dotted lines. Note that, in Fig. 2, due to size constraints, we only show the relevant parts of the time scale. As specified by the GCTT and its extensions, Fig. 2(a) assumes the aperiodic task is transformed into a periodic task (with period equivalent to the minimum inter-arrival time, 240), and the corresponding triggering event first arrives at the same time as the two other (periodic) tasks (i.e., time 0). In that case, assuming the target task is $t3$, the differences between the execution end and the deadline are 10 and 20 time units, for each of the two executions respectively: $d_{t3,1-}e_{t3,1} = (250 * 1) - 240$, $d_{t3,2-}e_{t3,2} = (250 * 2) - 480$.

If we use RTTT to generate the seeding times based on the data in Table 2, we obtain the timing diagram in Fig. 2(b). Notice that the second execution of the target task is now 10 time units closer to its deadline than with the GCTT assumptions ($t3$ starts executing at time unit 250, executes for five time units before it is preempted by $t1$, then resumes execution at time unit 475 and executes for 15 time units. Its deadline is 500.). The difference $d_{t3,1} - e_{t3,1}$ is unchanged. This illustrates that RTTT can be used to generate test cases that will stress the system more than the scenarios entailed by schedulability theory in situations where there are aperiodic tasks.

RTTT also reports that with only a 4.5% execution time estimate increase for each task (which would correspond to a very small error), a deadline miss appears in the target task, as illustrated in Fig. 2(c) (seeding times are the same as in Fig. 2(b), i.e., the output from RTTT, but the execution times are increased by 4.5%). The first executions of the three tasks end at time units 210, 231 and 252 respectively, thus resulting in a missed deadline for $t3$ (2 time units). This indicates that any inaccuracy greater than 4.5% in execution time estimates will result in missed deadlines during test execution.

When running multiple (i.e., 10) GA executions on this case study no variance was observed in the output. This is probably due to the relative simplicity of the task architecture. In all 10 executions, with a 500 time unit testing interval (T), the value of the objective function, and the largest difference between completion and deadline times were the same. The average execution time of each execution of RTTT was one minute.

Table 3 Task characteristics: Example 2

	Task $t1$	Task $t2$	Task $t3$
Period (periodic), or minimum inter-arrival time (aperiodic)	3	9	9
Priority	32	31	30
Execution time	1	3	2

4.2. A schedulable task can miss a deadline

Consider the three tasks $t1$ (periodic), $t2$ (aperiodic) and $t3$ (periodic) whose characteristics are shown in Table 3. Like the previous example, $t1$ has a higher priority than $t2$ and $t3$. Additionally, $t1$ and $t3$ are interdependent, i.e., one cannot begin execution before the other has fully completed its execution, as they share a common resource. Using GCTT and its extension we can prove that these three tasks are schedulable (see details in Appendix A), which is illustrated in the timing diagram of Fig. 3(a). This figure further assumes that the aperiodic task is ready to execute at time 0, just like the periodic tasks. In that case, $t1$ completes its first execution by time unit 1 (2 time units before its deadline) at which point $t2$ can begin its execution. Task $t2$ is preempted by the higher priority task $t1$ at time unit 3, but it is allowed to complete execution after $t1$ has completed (at time unit 4). Task $t2$'s first execution ends at time unit 5, long before its deadline at time unit 8. At this point (i.e., at time unit 5), $t3$ is now ready to execute and does so. Although the higher priority task $t1$ arrives at time unit 6 while $t3$ is executing, $t3$ is not preempted ($t1$ is dependent on $t3$). It is allowed to complete its execution, which ends at time unit 7, 2 units before its deadline. Thus, all three tasks meet their first deadlines.

In [5, 6, 16, 18, 19], the authors state that, for schedulability analysis, aperiodic tasks can be modeled as periodic tasks with periods equivalent to the minimum inter-arrival time. The authors claim that doing so assumes the worst-case scenario whereby aperiodic tasks

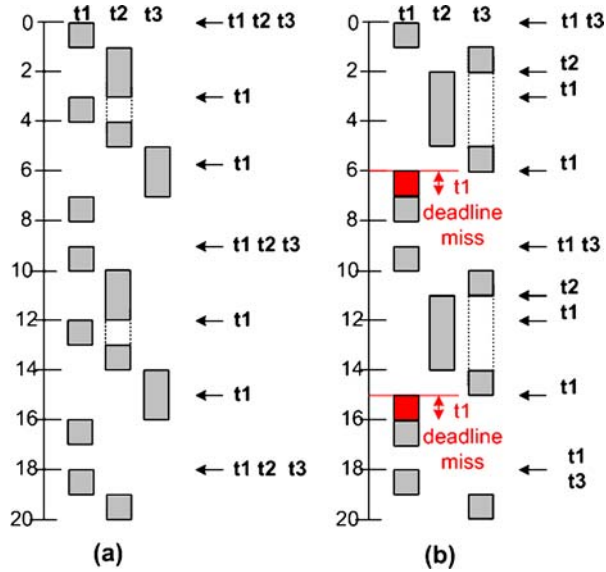

Fig. 3 Timing diagrams for Example 2: Schedulable tasks can miss deadlines

constantly arrive at their defined minimum inter-arrival times. Hence, any lower priority tasks will be preempted by the aperiodic tasks the maximum number of times possible, increasing the likelihood of deadline misses. However, this scenario does not accurately represent the worst-case scenario. The deadlines of lower priority tasks may be missed if the aperiodic task arrival is slightly shifted. Assume that the arrival times of aperiodic task $t2$ are slightly shifted to time units 2 and 11 respectively. These particular arrival times have been produced by our prototype tool. Because of its dependence on $t3$, $t1$ is no longer capable of meeting neither its second deadline at time unit 6, nor its fifth deadline at time unit 15 as illustrated in Fig. 3(b). It is important to note that these new arrival times comply with the minimum and maximum inter-arrival times of $t2$, the only aperiodic task, and they present a scenario that is worse than treating the aperiodic task as periodic. By triggering $t2$ at time unit 2, $t3$ is allowed to begin its execution for one time unit. When $t2$ preempts $t3$ and $t2$ begins its execution, $t1$ cannot pre-empt it at time unit 3 because of its dependency on $t3$. Hence, $t1$ can only begin its execution once $t3$ has completed its execution. However, $t3$ completes its execution at the deadline of $t1$, hence causing $t1$ to miss its deadline.

These results illustrate the fact that the GA based specification of performance stress test cases may lead to the automated identification of situations where deadlines are missed, even before testing is actually performed. This is the case for sets of tasks identified as schedulable by the GCTT.

The results of multiple runs of this case study also produced no variance in the output. In all 10 runs, the value of the objective function was 5.75 and the largest difference was 1 occurring in two executions: executions two and five of $t1$. For these three tasks running on a 20 time unit interval, the average execution time of each run was eight seconds.

4.3. An industrial case study

In an effort to demonstrate the feasibility of using predictable real-time scheduling technology and Ada in embedded systems, the Software Engineering Institute (SEI), the Naval Weapons Center and IBM's Federal Sector Division worked together to generate a hard real-time, realistic avionics application model. The endeavor was performed under the auspices of the SEI Real-Time Scheduling in Ada project. The joint effort proceeded by first defining the detailed performance and complexity requirements of a Generic Avionics Platform (GAP), similar to existing U.S. Navy and Marine aircrafts. The GAP task set is comprised of 18 tasks, the highest eight priorities of which are deemed schedulable according to schedulability analysis [10]. Only one task is aperiodic, `Weapon protocol` . All other tasks are periodic.

The three main tasks (highest priorities) in GAP are `Weapon Release, Weapon Aiming` and `Radar Tracking` . The weapon system is activated through an aperiodic event emulating a pilot button press requesting weapon release. The button press triggers the `Weapon Aiming` periodic task and waits for another button press to handle a possible abort request. Meanwhile, `Weapon Aiming` periodically computes the release time of the weapon (once every 50 ms). Throughout this time, the task constantly checks whether an abort request has been issued. Once one second remains to release, any abort request is denied and `Weapon Aiming` triggers the periodic `Weapon Release` task: This task has a 200 ms period and must complete its execution within five ms, i.e., 195 ms before the end of the current period. Once the release time is reached, `Weapon Release` proceeds to release one weapon every second for a total of five seconds. The interested reader is referred to [10] and [3].

Partial results (for four of the eight tasks) are presented in Table 4 (all results are available in Appendix B and presented in detail in [3]). The first column indicates the target task used

Table 4 Executing RTTT once for each task in the avionics application (partial results)

Target task t	$a_{\text{WeaponProtocol,j}}$	$e_{t,j} - d_{t,j}$ (ms)
`Weapon Release`	4887, 11919, 12122, 14749, 14990	–
`Radar Tracking Filter`	5, 207, 512, 870, 1197, 1587, 1827, 2213, 2444, 2660, 3024, 3249, 3495, 3852, 4159, 4373, 4658, 4870, 5070, 5393, 5593, 5801, 6065, 6304, 6553, 6784, 7040, 7249, 7453, 7669, 7900, 8110, 8488, 8859, 9075, 9459, 9668, 9915, 10183, 10472, 11083, 11422, 11811, 12219, 12441, 12789	–
`RWR Contact Management`	39, 321, 522, 738, 1090, 1341, 1574, 1945, 2335, 2541, 2754, 3143, 3346, 3660, 3989, 4300, 4544, 4841, 5050, 5302, 5671, 5875, 6091, 6440, 6803, 7025, 7260, 7518, 7760, 8114, 8360, 8724, 9021, 9333, 9593, 9915, 10132, 10350, 10570, 10938, 11308, 11552, 11754, 11966, 12221, 12535, 12769, 13089, 13372, 13572, 13856, 14090	3, 9
`Navigation Update`	157, 368, 579, 781, 998, 1377, 1589, 1816, 2043, 2317, 2593, 2829, 3036, 3376, 3595, 3861, 4108, 4408, 4633, 5003, 5214, 5419, 5624, 5858, 6125, 6333, 6536, 6787, 7174, 7408, 7611, 7833, 8199, 8588, 8859, 9142, 9356, 9615, 9850, 10060, 10309, 10550, 10814, 11111, 11415, 11669, 11877, 12097, 12325, 12552, 12784, 12995, 13208, 13419, 13666, 13914, 14119, 14336, 14536, 14739	1, 29, 23, 2, 28, 27, 32

during each execution of RTTT, with $T = 15$ s. The second column reports the arrival times that were generated by RTTT for the event triggering aperiodic task `Weapon Protocol` (the only aperiodic task), that maximize the objective function. The last column indicates the difference between completion time and deadline in those cases ($e_{t,j} - d_{t,j}$) when deadlines are missed. Two target tasks produced more than one deadline miss: two and seven deadline misses for tasks `RWR Contact Management` and `Navigation Update`, respectively (last column in Table 4).

Selecting the `weapon release` task as our target, RTTT produced no deadline misses. However, running RTTT once for each of the seven remaining highest priority tasks deemed schedulable, with each as target task, we were able to detect deadline misses. Out of the eight highest priority tasks, three result in deadline misses: `RWR Contact Management`, `Radar Target Update` and `Navigation Update` .`RWR`, or `Radar Warning Receiver` , is responsible for providing threat information from the environment surrounding the aircraft. `Radar Target Update` updates various target positions, while `Navigation Update` computes, for example, the aircraft position and altitude.

With the `Weapon Release` task as our target, RTTT further indicates that execution time estimates will not produce deadline misses if they are accurate within 22%. Executing RTTT with `Weapon Aiming` as a target task produces the same result. Considering that

task execution estimation is based on so many "guesses" and subject to so many factors (recall the discussion in Section 2.1), this value does not seem high enough to ensure that inaccuracies will not result at runtime, causing deadline misses. To be on the safe side, testers should then execute the test cases produced by RTTT for those two tasks during stress testing, in order to ensure that execution time inaccuracies do not lead to missing deadlines. On the other hand, for `Radar Tracking` , with a required error estimate of 1155%, a deadline miss at run-time is very unlikely.

Because any GA is a heuristic search algorithm, it is possible that some of the tasks that appear schedulable (i.e., do not show deadline misses in the execution of Table 4), could in fact miss deadlines. We investigate next whether, in practice, this is likely to be a problem.

The results of multiple GA executions of this case study produced some variance in the GA output, as shown in Table 5 (complete results are available in Appendix B and presented in detail in [3]: We executed RTTT 10 times with each task as a target task. With the first two tasks, `Weapon Release` and `Weapon Release Subtask` , the results of the 10 executions show very little variance and no deadline misses. The 10 executions of the GA, with `Weapon Release` as a target task, produce the same fitness function value (0) and largest difference $e_{t,j} - d_{t,j}$ (-197). Note that when the GA executes and returns a sequence of seeding times, more than one execution of the target task may be involved, and we return the largest value of $e_{t,j} - d_{t,j}$ for all the (j) executions of target task t. `Weapon Release Subtask` also conforms to its deadlines, though with some variance in the 10 executions: Executions 1, 2, 4, and 6 to 10 produce the same result, which is different from the third and fifth executions (Table 5). `Radar Tracking Filter` was additionally found to result in deadline misses in three of the 10 executions (positive values for $e_{t,j} - d_{t,j}$ in Table 5), e.g., one execution misses its deadline by 6 time units. Table 5 also shows that executing RTTT with `RWR Contact Management` as a target task systematically produced deadline misses with some variance (executions miss deadlines by 5 to 14 time units). In other words, each GA execution leads to at least one deadline miss. Additional results can be found in [3]. `Data Bus Poll Device` misses deadlines in two of the 10 executions, `weapon Aiming` always conforms to its deadlines, though with more variance than `Weapon Release Subtask` (the 10 executions produce different results). `Radar Target Update` and `Navigation Update` systematically miss deadlines within some variance.

In Table 5, the maximum difference in terms of largest value for $e_{t,j} - d_{t,j}$ appears to be 11 ms and 6 ms for the last two tasks, which both have a period of 25 ms. There is an important difference between `Radar Tracking Filter` and `RWR Contact Management` : the former task misses its deadlines only for a subset of executions whereas the latter task systematically misses all deadlines. This means that by executing the GA once, deadline misses could have gone unnoticed in the former. Therefore, we would advise that RTTT be executed several times for each target task, to increase chances of deadline misses detection. The exact number of runs will depend on GA execution times and the user's time constraints. From a practical standpoint, this is admissible when considering that the execution time for each task is quite reasonable. For the eight tasks of this case study, with $T = 15$ s, the average execution time of each GA execution was 46.5 min.[3] This execution time can be drastically decreased by using a newer, faster computer and by parallelizing the search [11]. To conclude, we do not expect the variability inherent to our GA to be an impediment in its practical use.

[3] A longer T leads to longer execution times.

Table 5 Executing RTTT 10 times for each task in the avionics application (partial results)

Target task t	Executions	Fitness function	Largest value of $e_{t,j} - d_{t,j}$
`Weapon Release`	1–10	0	−197
`Weapon Release Subtask`	1,2,4,6–10	0	−997
	3	0	−994
	5	0	−995
`Radar Tracking Filter`	1	0.191298	−3
	2	0.116635	−4
	3	64.002	6
	4	2.00288	1
	5	0.0889019	−4
	6	0.0427746	−5
	7	0.559302	−1
	8	0.0685323	−4
	9	18.0067	4
	10	0.130897	−3
`RWR Contact Management`	1	520.035	9
	2	112.96	5
	3	32.094	5
	4	552.665	8
	5	2049.11	11
	6	16664.2	14
	7	8640.07	13
	8	136.587	7
	9	403.03	8
	10	162.134	7

5. Conclusion

Reactive real-time systems have to react to external events within time constraints: Triggered tasks must execute within deadlines. The goal of this article is to automate, based on the system task architecture, the derivation of test cases; that is seeding times for aperiodic tasks that maximize the chances of critical deadline misses.

Deadlines may be missed even though the associated tasks have been identified as schedulable through appropriate schedulability analysis (the Generalized Completion Time Theorem—GCTT). The main reason for this is that the Generalized Completion Time Theorem makes a number of simplifying assumptions when dealing with aperiodic tasks: aperiodic tasks are modeled as periodic tasks with periods equivalent to the minimum inter-arrival time; aperiodic tasks are ready to execute at time unit 0, like periodic tasks.

Our automation tailors Genetic Algorithms to address the automated generation of seeding times for aperiodic tasks based on task information, such as estimated execution times and priorities. In other words, for a given group of tasks, times are identified for external events to which the system is supposed to react in order to maximize the chances of missed deadlines. Users are free to focus on tasks they deem critical for the application.

We have performed a number of case studies using our tool, RTTT, varying the number of tasks and priority patterns in each. We came across a number of cases that suggest that

RTTT can identify seeding times that stress the system to such an extent that small errors in the execution time estimates can lead to missed deadlines during stress testing.

Another practical result is that RTTT can identify cases where, even when tasks have been determined to be schedulable by theoretical means, seeding times can lead to missed deadlines. This is due to the underlying assumptions of GCTT. This suggests that both techniques (schedulability theory and GA-based stress testing) be applied early during the design of real-time systems. GCTT should be used as a first schedulability check and then, in the presence of aperiodic tasks, if all tasks are deemed schedulable, RTTT should be used to further check the most critical tasks.

The case study in this paper, which is based on an actual, pre-existing system task architecture, confirmed that RTTT could be useful in practice. Indeed, it proved to be very helpful in identifying performance problems in the early design stages, once there is a defined task architecture specifying estimated execution times and priorities, and before any running implementation is available. Furthermore, our empirical results suggest that the inherent variance of results across GA executions and their execution times are not an impediment to the practical use of our approach.

In future work, we will first improve our Genetic Algorithm and use an adaptive termination criterion [12, 23] instead of a fixed number of generations. We will also try to improve the execution time of our algorithms. Another important improvement will be to account for events' parameters in our chromosome, although the impact of parameters is usually accounted for when estimating task executions. Last, our approach needs more extensive validation on additional (industrial) case studies.

Appendix A: Schedulability analysis for Example 2

The Generalized Completion Time Theorem (GCTT) determines whether periodic tasks are schedulable, i.e., whether tasks can meet their deadlines, given preemption by higher priority tasks and blocking time by lower priority tasks. First note that this applies only to periodic tasks. However, for schedulability analysis, aperiodic tasks are considered equivalent to periodic tasks whose periods are equal to the minimum inter-arrival times [5]. The theorem assumes all tasks are ready for execution at the start of the task's period: i.e., at time zero, all tasks are awaiting execution [16]. It is important to note that the GCTT can only determine whether a group of tasks is schedulable. It cannot show whether they are not schedulable. (Hence, if the GCTT check fails, the group of tasks at hand may still be schedulable.) Additionally, it is argued [16] that "when all the tasks are initiated at the same time (the worst-case phasing), if a task completes its execution before the end of its first period, it will never miss a deadline."

As discussed before, GCTT can be graphically illustrated by a timing diagram [5]. We rather focus here on its mathematical formulation.

A set of n periodic tasks can be scheduled by the Rate Monotonic Algorithm[4] for all tasks phasing if the following condition is satisfied:

$$\forall i, 1 \leq i \leq n, \min_{(k,l)\in R_i} \left(\sum_{j=1}^{i-1} \frac{C_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil + \frac{C_j}{lT_k} + \frac{B_i}{lT_k} \right) \leq 1$$

[4] The Rate Monotonic Algorithm is extended to deal with priority inversion through the Generalized Completion Time Theorem.

where

$$R_i = \left\{ (k, l) \,|\, 1 \leq k \leq i, l = 1, \ldots, \left\lfloor \frac{T_i}{T_k} \right\rfloor \right\}.$$

(C_i and T_j are the execution time and period of task i. B_i is the worst-case blocking time for task i)[5]

The theorem checks if each task can complete its execution before its first deadline, by checking the amount of time spent waiting for higher priority tasks (preemption) and lower dependent priority tasks (blocking). In the formula, i denotes the task to be checked and k denotes each of the tasks that affects the completion time of task i, i.e., task i and the higher priority tasks. For a given i and k, each value of l represents the number of times task k can execute during task i's period, which corresponds to the amount of time during which task i is preempted. Note that when determining B_i, each task on which i depends can block i only once because of priority differences.

Let us apply this theorem to the example in Section 4.2. The task architecture is recalled here to ease reading: Task $t1$ is periodic ($C_1 = 1$, $T_1 = 3$); Task $t2$ is aperiodic but transformed into a periodic task for schedulability analysis purposes ($C_2 = 3$, $T_2 = 9$); Task $t3$ is periodic ($C_3 = 2$, $T_3 = 9$); $t1$ and $t3$ are inter-dependent; $t1$ has the highest priority, $t2$ has the second highest priority and $t3$ has the lowest priority.

Let us apply the theorem on each of the three tasks in sequence.

Task $t1$

$$R_1 = \left\{ (k, l) | 1 \leq k \leq 1, l = 1, \ldots, \left\lfloor \frac{T_1}{T_k} \right\rfloor \right\} = \{(1, 1)\}.$$

Then,

$$\min_{(k,1)\in R_1} \left(\sum_{j=1}^{0} \frac{C_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil + \frac{C_1}{lT_k} + \frac{B_1}{lT_k} \right)$$

$$= \min_{(1,1)} \left(\frac{C_1}{lT_k} + \frac{B_1}{lT_k} \right) = \frac{C_1}{T_1} + \frac{B_1}{T_1} = \frac{1}{3} + \frac{2}{3} = \frac{3}{3} \leq 1.$$

Indeed, $t1$ can only be blocked by lower priority task $t3$ once during a period of 2 time units: $B_1 = 2$. This means that $t1$ meets its first deadline.

Task $t2$

$$R_2 = \left\{ (k, l) | 1 \leq k \leq 2, l = 1, \ldots, \left\lfloor \frac{T_2}{T_k} \right\rfloor \right\}.$$

For $k = 1, 2$, that is, $l = 1, 2, 3$. For $k = 2, l = 1, \ldots, \lfloor \frac{T_2}{T_2} \rfloor$ that is, $l = 1$. Thus $R_2 = \{(1, 1), (1, 2), (1, 3), (2, 1)\}$. Note that since $t2$ is not dependent on the other two tasks,

[5] Note that this mathematical formulation of the problem assumes tasks are ordered in decreasing order of priority (for the purpose of schedulability analysis), i.e., task numbered 1 has the highest priority, task numbered 2 has the second highest priority, . . ., task numbered n has the lowest priority.

especially $t3$ (lowest priority task), $t2$ cannot be blocked by $t3$ and then $B_2 = 0$.

$$\text{For } (k,l) = (1,1), \sum_{j=1}^{1} \frac{C_j}{T_1}\left\lceil \frac{T_1}{T_j} \right\rceil + \frac{C_2}{T_1} + \frac{B_2}{T_1} = \frac{C_1}{T_1}\left\lceil \frac{T_1}{T_1} \right\rceil + \frac{C_2}{T_1}$$
$$= \frac{C_1}{T_1} + \frac{C_2}{T_1} = \frac{1}{3} + \frac{3}{3} > 1.$$

$$\text{For } (k,l) = (1,2), \sum_{j=1}^{1} \frac{C_j}{2T_1}\left\lceil \frac{2T_1}{T_j} \right\rceil + \frac{C_2}{2T_1} + \frac{B_2}{2T_1} = \frac{C_1}{2T_1}\left\lceil \frac{2T_1}{T_1} \right\rceil + \frac{C_2}{2T_1}$$
$$= \frac{2C_1}{2T_1} + \frac{C_2}{2T_1} = \frac{2}{6} + \frac{3}{6} \leq 1.$$

$$\text{For } (k,l) = (1,3), \sum_{j=1}^{1} \frac{C_j}{3T_1}\left\lceil \frac{3T_1}{T_j} \right\rceil + \frac{C_2}{3T_1} + \frac{B_2}{3T_1} = \frac{C_1}{3T_1}\left\lceil \frac{3T_1}{T_1} \right\rceil + \frac{C_2}{3T_1}$$
$$= \frac{3C_1}{3T_1} + \frac{C_2}{3T_1} = \frac{3}{9} + \frac{3}{9} \leq 1.$$

$$\text{For } (k,l) = (2,1), \sum_{j=1}^{1} \frac{C_J}{T_2}\left\lceil \frac{T_2}{T_j} \right\rceil + \frac{C_2}{T_2} + \frac{B_2}{T_2} = \frac{C_1}{T_2}\left\lceil \frac{T_2}{T_1} \right\rceil + \frac{C_2}{T_2}$$
$$= \frac{3C_1}{T_2} + \frac{C_2}{T_2} = \frac{3}{8} + \frac{3}{8} \leq 1.$$

This means that $t2$ meets its first deadline.[6]

Task *t3*

$$R_3 = \left\{ (k,l) | 1 \leq k \leq 3, l = 1, \ldots, \left\lfloor \frac{T_3}{T_k} \right\rfloor \right\}.$$

For $k = 1$, $l = 1, \ldots, \lfloor \frac{T_3}{T_1} \rfloor$, that is, $l = 1, 2, 3$. For $k = 2$, $l = 1, \ldots, \lfloor \frac{T_3}{T_2} \rfloor$, that is, $l = 1$. For $k = 3, l = 1, \ldots, \lfloor \frac{T_3}{T_3} \rfloor$, that is $l = 1$. Since there is no lower priority task than $t3, B_3 = 0$.

$$\text{For } (k,l) = (1,1), \sum_{j=1}^{2} \frac{C_j}{T_1}\left\lceil \frac{T_1}{T_j} \right\rceil + \frac{C_3}{T_1} + \frac{B_3}{T_1} = \frac{C_1}{T_1}\left\lceil \frac{T_1}{T_1} \right\rceil + \frac{C_2}{T_1}\left\lceil \frac{T_1}{T_2} \right\rceil$$

[6] Note that since we are looking for a minimum, in a set of sums, that is below or equal to 1, we can stop our investigation of elements (k, l) in R_i as soon as we find a pair for which the sum is indeed below or equal to 1: if this is the minimum we can stop, if not this ensures that the minimum (necessarily below the value we find below or equal to 1) is indeed below or equal to 1. For tasks $t2$ and $t3$, we can thus stop at $(k,l) = (1,2)$ and $(k,l) = (2,1)$, respectively.

$$+\frac{C_3}{T_1}+\frac{B_3}{T_1}=\frac{C_1}{T_1}+\frac{C_2}{T_1}+\frac{C_3}{T_1}+\frac{B_3}{T_1}=\frac{6}{3}>1.$$

$$\begin{aligned}\text{For}\,(k,l)=(1,2),\sum_{j=1}^{2}\frac{C_j}{2T_1}\left\lceil\frac{2T_1}{T_j}\right\rceil+\frac{C_3}{2T_1}+\frac{B_3}{2T_1}\\ =\frac{C_1}{2T_1}\left\lceil\frac{2T_1}{T_1}\right\rceil+\frac{C_2}{2T_1}\left\lceil\frac{2T_1}{T_2}\right\rceil+\frac{C_3}{2T_1}+\frac{B_3}{2T_1}\\ =\frac{2C_1}{2T_1}+\frac{C_2}{2T_1}+\frac{C_3}{2T_1}+\frac{B_3}{2T_1}=\frac{7}{6}>1.\end{aligned}$$

$$\begin{aligned}\text{For}\,(k,l)=(1,3),\sum_{j=1}^{2}\frac{C_j}{3T_1}\left\lceil\frac{3T_1}{T_j}\right\rceil+\frac{C_3}{3T_1}+\frac{B_3}{3T_1}\\ =\frac{C_1}{3T_1}\left\lceil\frac{3T_1}{T_1}\right\rceil+\frac{C_2}{3T_1}\left\lceil\frac{3T_1}{T_2}\right\rceil+\frac{C_3}{3T_1}+\frac{B_3}{3T_1}\\ =\frac{3C_1}{3T_1}+\frac{2C_2}{3T_1}+\frac{C_3}{3T_1}+\frac{B_3}{3T_1}=\frac{11}{9}>1.\end{aligned}$$

$$\begin{aligned}\text{For}\,(k,l)=(2,1),\sum_{j=1}^{2}\frac{C_J}{T_2}\left\lceil\frac{T_2}{T_j}\right\rceil+\frac{C_3}{T_2}+\frac{B_3}{T_2}=\frac{C_1}{T_2}\left\lceil\frac{T_2}{T_1}\right\rceil+\frac{C_2}{T_2}\left\lceil\frac{T_2}{T_2}\right\rceil+\frac{C_3}{T_2}+\frac{B_3}{T_2}\\ =\frac{3C_1}{T_2}+\frac{C_2}{T_2}+\frac{C_3}{T_2}+\frac{B_3}{T_2}=\frac{8}{9}\leq 1.\end{aligned}$$

$$\begin{aligned}\text{For}\,(k,l)=(3,1),\sum_{j=1}^{2}\frac{C_j}{T_3}\left\lceil\frac{T_3}{T_j}\right\rceil+\frac{C_3}{T_3}+\frac{B_3}{T_3}=\frac{C_1}{T_3}\left\lceil\frac{T_3}{T_1}\right\rceil+\frac{C_2}{T_3}\left\lceil\frac{T_3}{T_2}\right\rceil+\frac{C_3}{T_3}+\frac{B_3}{T_3}\\ =\frac{3C_1}{T_3}+\frac{2C_2}{T_3}+\frac{C_3}{T_3}+\frac{B_3}{T_3}=\frac{11}{9}>1.\end{aligned}$$

This means that $t3$ meets its first deadline.[6]

Conclusion

As a conclusion, all three tasks meet their first deadlines, which demonstrates their schedulability.

Appendix B: Additional results for the industrial case study

Tables 6 and 7 below show the complete results of executing RTTT for each task of the Avionics case study. Results in Table 6 are partially presented in Table 4 whereas results in Table 7 are partially presented in Table 5.

Table 6. Executing RTTT once for each task in the avionics application (complete results)

Target task t	$a_{\text{Weapon Protocol},j}$	$e_{t,j} - d_{t,j}$ (ms)	Estimation error (%)
`Weapon Release`	4887, 11919, 12122, 14749, 14990	–	22.2
`Weapon Release Subtask`	9999, 14066, 14630, 14947		22.2
`Radar Tracking Filter`	5, 207, 512, 870, 1197, 1587, 1827, 2213, 2444, 2660, 3024, 3249, 3495, 3852, 4159, 4373, 4658, 4870, 5070, 5393, 5593, 5801, 6065, 6304, 6553, 6784, 7040, 7249, 7453, 7669, 7900, 8110, 8488, 8859, 9075, 9459, 9668, 9915, 10183, 10472, 11083, 11422, 11811, 12219, 12441, 12789	—	1155.5
`RWR Contact Management`	39, 321, 522, 738, 1090, 1341, 1574, 1945, 2335, 2541, 2754, 3143, 3346, 3660, 3989, 4300, 4544, 4841, 5050, 5302, 5671, 5875, 6091, 6440, 6803, 7025, 7260, 7518, 7760, 8114, 8360, 8724, 9021, 9333, 9593, 9915, 10132, 10350, 10570, 10938, 11308, 11552, 11754, 11966, 12221, 12535, 12769, 13089, 13372, 13572, 13856, 14090	3, 9	0
`Data Bus Poll Device`	90, 299, 601, 892, 1093, 1295, 1687, 1896, 2100, 2415, 2802, 3048, 3261, 3476, 3684, 3893, 4094, 4294, 4509, 4730, 4931, 5180, 5440, 5797, 6015, 6244, 6460, 6669, 6884, 7093, 7308, 7525, 7790, 8014, 8343, 8737, 9099, 9330, 9533, 9850, 10051, 10263, 10556, 10756, 10969, 11174, 11887, 12693, 13274, 14981	–	211.1
`Weapon Aiming`	94, 302, 508, 748, 1123, 1396, 1626, 1882, 2098, 2371, 2580, 2888, 3094, 3294, 3676, 3945, 4145, 4347, 4733, 5021, 5386, 5774, 6120, 6382, 6594, 6799, 7011, 7389, 7596, 7922, 8179, 8387, 8592, 9135, 9650, 10153, 10354, 10571, 10780, 11077, 11362, 12008, 12210, 12436, 12825, 13092, 13316, 13550	–	22.2
`Radar Target Update`	107, 433, 653, 858, 1241, 1460, 1674, 2099, 2343, 2678, 2897, 3115, 3319, 3581, 3917, 4118, 4318, 4525, 4726, 5046, 5284, 5557, 5795, 6001, 6301, 6506, 6707, 6960, 7269, 7536, 7888, 8139, 8380, 8599, 8858, 9233, 9537, 9874, 10078, 10282, 10546, 10758, 11013, 11324, 11530, 11792, 11999, 12202, 12572, 12826, 13026, 13257	17, 16, 10, 9	0
`Navigation Update`	157, 368, 579, 781, 998, 1377, 1589, 1816, 2043, 2317, 2593, 2829, 3036, 3376, 3595, 3861, 4108, 4408, 4633, 5003, 5214, 5419, 5624, 5858, 6125, 6333, 6536, 6787, 7174, 7408, 7611, 7833, 8199, 8588, 8859, 9142, 9356, 9615, 9850, 10060, 10309, 10550, 10814, 11111, 11415, 11669, 11877, 12097, 12325, 12552, 12784, 12995, 13208, 13419, 13666, 13914, 14119, 14336, 14536, 14739	1, 29, 23, 2, 28, 27, 32	0

Table 7 Executing RTTT 10 times for each task in the avionics application (complete results)

Target task t	Run number	Objective function value	Largest value of $e_{t,j} - d_T$
`Weapon Release`	1–10	0	–197
`Weapon Release Subtask`	1,2,4,6–10	0	–997
	3	0	–994
	5	0	–995
`Radar Tracking Filter`	1	0.191298	–3
	2	0.116635	–4
	3	64.002	6
	4	2.00288	1
	5	0.0889019	–4
	6	0.0427746	–5
	7	0.559302	–1
	8	0.0685323	–4
	9	18.0067	4
	10	0.130897	–3
`RWR Contact Management`	1	520.035	9
	2	112.96	5
	3	32.094	5
	4	552.665	8
	5	2049.11	11
	6	16664.2	14
	7	8640.07	13
	8	136.587	7
	9	403.03	8
	10	162.134	7
`Data Bus Poll Device`	1	0.0371494	–5
	2	0.562528	–1
	3	0.257822	–2
	4	2.01564	1
	5	0.0629904	–4
	6	0.039091	–5
	7	0.00172283	–12
	8	0.000150355	–14
	9	16.0001	4
	10	0.500017	–1
`Weapon Aiming`	1	0.0201419	–6
	2	0.00586035	–8
	3	0.00595128	–8
	4	0.126953	–3
	5	0.000214935	–13
	6	0.0195313	–6
	7	0.000489237	–11
	8	0.157288	–3
	9	0.0312508	–5
	10	0.00394637	–8

(Continued on next page)

Table 7 Continued

Target task t	Run number	Objective function value	Largest value of $e_{T,j} - d_T$
`Radar Target Update`	1	198144	17
	2	1058300	20
	3	1048580	20
	4	139265	17
	5	140288	17
	6	262146	18
	7	590081	19
	8	163840	17
	9	81921.2	16
	10	24578.3	14
`Navigation Update`	1	5242880000	32
	2	6442450000	32
	3	18387800000	34
	4	17179900000	34
	5	67148000	26
	6	138513000000	37
	7	72351700	26
	8	146029000000	37
	9	18547300000	34
	10	309240000000	38

Acknowledgments This work was partly supported by a Canada Research Chair (CRC) grant. Lionel Briand and Yvan Labiche were further supported by NSERC operational grants. This work is part of a larger project on testing object-oriented systems with the UML (www.sce.carleton.ca/Squall).

References

1. B. Beizer, *Software Testing Techniques*, 2nd edition, Van Nostrand Reinhold, 1990.
2. R. V. Binder, *Testing Object-Oriented Systems—Models, Patterns, and Tools*, Object Technology, Addison-Wesley, 1999.
3. L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing for real-time systems using genetic algorithms," Carleton University, Technical Report SCE-03-23, http://www.sce.carleton.ca/Squall/, September 2003.
4. A. E. Eiben, P. E. Raue, and Z. Ruttkay, "Solving constraint satisfaction problems using genetic algorithms," in *Proc. IEEE World Conference on Evolutionary Computing*, 1994, pp. 542–547.
5. H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Object Technology, Addison Wesley, 2000.
6. M. G. Harbour, M. H. Klein, R. Obenza, B. Pollak, and T. Ralya, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*, Kluwer, 1993.
7. R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*, Wiley-Interscience, 1998.
8. M. A. Iverson, F. Ozguner, and L. C. Potter, "Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment," *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1374–1379, 1999.
9. J. W. S. Liu, *Real-Time Systems*, Prentice Hall, 2000.
10. C. D. Locke, D. R. Vogel, and T. J. Mesler, "Building a predictable avionics platform in ada: A case study," in *Proc. IEEE Real Time Systems Symposium*, 1991, pp. 181–189.
11. S. W. Mahfoud and D. E. Goldberg, "Parallel recombinative simulated annealing: A genetic Algorithm," *Parallel Computing*, vol. 21, no. 1, pp. 1–28, 1995.
12. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer, 1996.

13. F. Mueller and J. Wegener, "A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints," in *Proc. IEEE Real Time Technology and Applications Symposium*, 1998, pp. 179–188.
14. M. A. Pawlowsky, "Crossover Operators," in *Practical Handbook of Genetic Algorithms Applications*, L. Chambers (ed.), CRC Press, 1995, vol. 1, pp. 101–114.
15. J. D. Schaffer, R. A. Caruna, L. J. Eshelman, and R. Das, "A study of control parameters affecting online performance of genetic algorithms for function optimization," in *Proc. International Conference on Genetic Algorithms and Their Applications*, 1989, pp. 51–60.
16. L. Sha and J. B. Goodenough, "Real-time scheduling theory and ada," Software Engineering Institute, Technical Report CMU/SEI-89-TR-014, 1989.
17. J. E. Smith and T. C. Fogarty, "Adaptively parameterized evolutionary systems: Self adaptive recombination and mutation in a genetic algorithm," in *Parallel Problem Solving From Nature*,Voigt, Ebeling, Rechenberg, and Schwefel, (eds.), 1996, vol. 4, pp. 441–450.
18. B. Sprunt, *Aperiodic Task Scheduling for Real-Time Systems*, Ph.D. Thesis, Carnegie Mellon, Department of Electrical and Computer Engineering, 1990.
19. B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-Time Systems*, vol. 11, pp. 27–60, 1989.
20. N. Tracey, J. A. Clark, K. C. Mander, and J. A. McDermid, "An automated framework for structural test-data generation," *Proc. IEEE Conference on Automated Software Engineering*, 1989, pp. 285–288.
21. M. Wall, "GAlib: A C++ Library of Genetic Algorithm Components," Massachusetts Institute of Technology, http://lancet.mit.edu/ga/dist/galibdoc.pdf, August, 1996.
22. J. Zhang and S. C. Cheung, "Automated test case generation for the stress testing of multimedia systems," *Software—Practice and Experience*, vol. 32 15, pp. 1411–1435, 2002.
23. B. J. Jain, H. Pohlheim, and J. Wegener, "On termination criteria of evolutionary algorithms," in *Proc. Genetic and Evolutionary Computation Conference*, 2001, p. 768.