Stress Testing Real-Time Systems with Genetic Algorithms

Lionel C. Briand　Yvan Labiche　Marwa Shousha

Software Quality Engineering Laboratory
Department of Systems and Computer Engineering
Carleton University, 1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada

(613) 520 2600 ext. 2471　(613) 520 2600 ext. 5583

briand@sce.carleton.ca　labiche@sce.carleton.ca　mshousha@yahoo.com

ABSTRACT

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 maximize the chances of critical deadline misses within the system. We refer to that testing activity as stress testing. We have developed a method based on genetic algorithms and implemented it in a tool. Case studies were run and results show that the tool may actually help testers identify test cases that will likely stress the system to such an extent that some tasks may miss deadlines.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – validation, reliability

General Terms

Algorithms, Performance, Design, Reliability, Experimentation, Theory, Verification.

Keywords

Schedulability theory, Genetic algorithms

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 [4]. 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 [4]. 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 [4, 14] 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 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 [16].

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 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* since this is defined as "subjecting the system to harsh inputs […] with the intention of breaking it" [1]. Similarly, "a stress test pushes [the system] beyond its design limits. It is designed to cause a failure" [2], and the inability to meet a deadline is no less a fault than erroneous outputs.

The stress testing strategy presented in this paper consists in finding combinations of inputs such that completion times of a specific task's executions are as close as possible to their deadlines. Those combination of inputs, i.e., test cases, should account for both seeding times for aperiodic tasks and their possible input data 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 [7]. Finding such test cases is not an easy problem to solve as many periodic and aperiodic tasks, with different priorities, can be triggered. But, if a practical way to automate it is found, it would allow us to specify a scenario of event arrival times that makes it likely for tasks to miss their deadlines if their execution time estimates turn out, once the system is implemented, to be too inaccurate.

The search for an optimal combination of inputs uses a Genetic Algorithm (GA), as the problem to be solved is an optimization (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 a good chance of being schedulable. Stress testing can then be planned early and, once the implementation is available and after completion of functional testing, stress testing may begin right away.

GECCO'05, June 25-29, 2005, Washington, DC, USA.

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 solve our problem. 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, e.g., the round robin algorithm uses a FIFO list of tasks, while other algorithms use task priorities and preemption [4, 16]. The Portable Operating System Interface Standard (POSIX), to which most commercial real-time operating systems conform, assumes fixed pre-emptive 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 (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, that are ready to execute when the system starts executing and that are triggered at regular time intervals as determined by their minimum inter-arrival times [4, 14, 16]. We do not further discuss these techniques and refer the interested reader to [4, 14, 16].

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) [7]. At design time, analytic benchmarking and code profiling are the most common strategies employed to determine those estimates [6]. Analytic benchmarking consists in measuring the performance of a selected hardware when representative code samples (primitive code types) are executed. This information is 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 [4].) 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 [6]). 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., [17]).

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 [11]. 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 those 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 those 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 [19] 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. They are determined using linear programming 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, and our focus is on missing deadlines as opposed to resource saturation.

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

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

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 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) [5]. 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 [9]. 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. $A_{i,j}$ and $P_{i,j}$ denote the j^{th} 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 in increasing order 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 $a_{i,j-1}$ as well as min_i and max_i (if max_i is not specified, its value is set to T). 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

[2] This target task can be the most 'critical' task in the system, for instance. This analysis can be repeated for all 'critical' tasks, leading to several test scenarios, each focusing on stress testing a specific task.

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 [5]. 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.

3.3.1 *Cross-over operator*

To avoid mixing tasks executions and violating constraints too often, our cross-over operator is an n-point cross-over [12] 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 $k_{t1} + k_{t2} + \ldots + k_{tn-1}$ 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.

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)

3.3.2 *Mutation Operator*

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 $a'_{i,j}$ in a way similar to the initialization of the population for subsequent executions of the same task that no longer uphold the inter-arrival time constraints. Furthermore, if after mutation, the difference between the last arrival time of j and T is greater than max_i, then there is room for an additional arrival time and a gene with seeding time –1 is modified to fill the gap. 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 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} = 400), 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 would have required that gene 5's arrival time be changed.

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. Its 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 execution's 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.

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 [5], 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 [5]. 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 [5].

Throughout the GA literature, various mutation rates, crossover rates, and population sizes have been used [5]. 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 [15]. We thus apply the mutation rate suggested in [5], i.e., $\frac{1.75}{\lambda\sqrt{l}}$, where λ denotes the population size and *l* is the length of the chromosome [13]. Similarly, consistent with the observations reported in [5], 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 [10]: (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. Alternative, 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.

It is important to note that the technique we describe is readily applicable. A variety of software tools and frameworks exist that manage GAs, allowing users to perform optimization in a variety of programming languages using different representation and genetic operators. One can easily use them and apply our approach.

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 task, the task information, comprised of a task number, priority, estimated execution time, dependencies to other tasks (if any), a period in case of a periodic task or minimum and (possibly) maximum inter-arrival times and deadline in case of an aperiodic task; (2) test environment information comprised of the time interval during which the test is to be performed and the target task; and (3) whether the tool should output a timing diagram corresponding to the result.

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 Figure 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. RTTT assumes inaccuracies to be the same for all tasks. Though this is not likely to be the case in practice, this is a necessary simplifying assumption. The third result is timing diagrams of both the outputted GA solution as well as the safe estimate percentage.

```
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
```

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

The RTTT prototype executes on those inputs 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 [18]. The second module is a POSIX [4] 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 pre-emptive priority scheduler assuming single processor execution. Task dependencies in our application are in the form of shared resource dependencies [4]. 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. Case Studies

This section presents two 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). In the second example (Section 4.2) we show the usefulness and feasibility of the approach on an actual real-time system whose task architecture is public domain [8].

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 objective function and difference between execution end and deadline. Average execution times are also reported for each case study, running on an 800MHz AMD Duron processor with 192KB on-chip cache memory and 64MB 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. Tasks t1 and t3 are interdependent: Hence, 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 (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 .a). The timing diagram notation we use is an adaptation of the one proposed in [4]: 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 with dotted lines. Note that, in , due to size constraints, we only show the relevant parts of the time scale. As specified by the GCTT and its extensions, .a assumes the aperiodic task is transformed into a periodic task (with period equivalent to the minimum inter-arrival time, 8) 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.

Table 2. Task characteristics, an example

	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

If we use RTTT to generate the seeding times based on the data in Table 2, we obtain the timing diagram in .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 .c (seeding times are the same as in .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

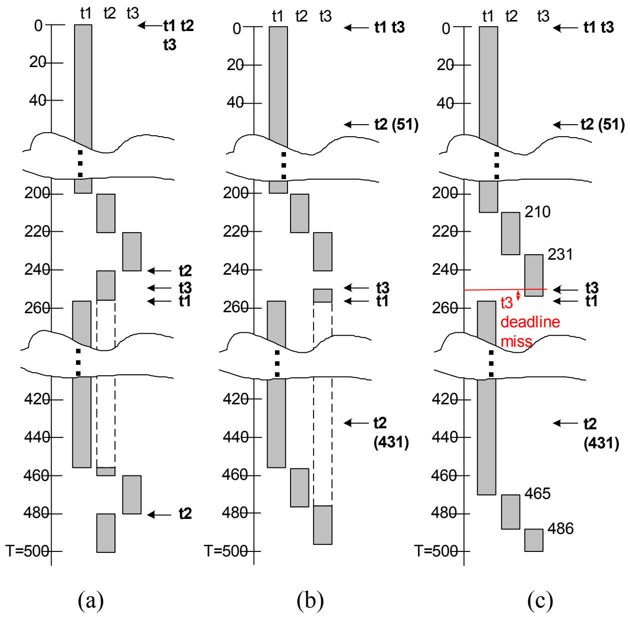

Figure 2. Execution times must be accurate, an example.

completion and deadline times were the same. The average execution time of each execution of RTTT was one minute.

4.2 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 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 [8]. 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 [8] and [3].

With the `Weapon Release` task as our target, RTTT 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 into 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.

For the eight (schedulable) tasks of this case study, assuming T =15s, when selecting `Weapon Release` as a target task, the value of the objective function and the largest difference between completion and deadline times were the same. However, for `Radar Tracking` some significant variance is observed and results suggest that RTTT should be run a minimum of 10 times to obtain better stress test cases [3]. From a general perspective, it is not possible to decide beforehand how many GA runs are necessary to ensure that near-optimal results are obtained. In practice, the user can only decide the number of runs based on available time and observed execution performance. The average execution time of each GA execution was 46.5 minutes. It is important to note that a longer T would lead to longer execution times, and that this execution time can be drastically decreased by using a newer, faster computer and by parallelizing the search [9].

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 maximize the chances of critical deadline misses.

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, regardless of the tasks at hand, times are identified for external events to which the system is supposed to react in order to maximize the chances of exhibiting 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, and 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. Because of underlying assumptions regarding aperiodic tasks, such results are obtained even when the set of tasks at hand have been proven schedulable by theoretical means (e.g., the Generalized Completion Time Theorem – GCTT), thus suggesting 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.

To conclude, this paper provides a practical solution that can help automate the stress testing of real-time, reactive systems. This automation is likely to help testers identify response time problems, either during design or testing. Another result, suggested by our studies [3] but not reported here, as we focused on stress testing, is 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. This is due to the assumptions of theorems such as

GCTT, and this may be very helpful in identifying performance problems in the early design stages.

In future work, we will first improve our Genetic Algorithm and use an adaptive termination criterion [10] 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.

6. ACKNOWLEDGEMENTS

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).

7. REFERENCES

[1] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 2nd Edition, 1990.

[2] R. V. Binder, *Testing Object-Oriented Systems - Models, Patterns, and Tools*, 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] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison Wesley, 2000.

[5] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*, Wiley, 1998.

[6] 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 (12), pp. 1374-1379, 1999.

[7] J. W. S. Liu, *Real-Time Systems*, Prentice Hall, 2000.

[8] C. D. Locke, D. R. Vogel and T. J. Mesler, "Building a predictable avionics platform in ada: A case study," *Proc. IEEE Real Time Systems Symposium*, pp. 181-189, 1991.

[9] S. W. Mahfoud and D. E. Goldberg, "Parallel Recombinative Simulated Annealing: A Genetic Algorithm," *Parallel Computing*, vol. 21 (1), pp. 1-28, 1995.

[10] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer, 1996.

[11] F. Mueller and J. Wegener, "A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints," *Proc. IEEE Real Time Technology and Applications Symposium*, pp. 179-188, 1998.

[12] M. A. Pawlowsky, "Crossover Operators," in L. Chambers, Ed., *Practical Handbook of Genetic Algorithms Applications*, vol. 1, CRC Press, pp. 101-114, 1995.

[13] 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," *Proc. International Conference on Genetic Algorithms and Their Applications*, pp. 51-60, 1989.

[14] L. Sha and J. B. Goodenough, "Real-time Scheduling Theory and Ada," Software Engineering Institute, Technical Report CMU/SEI-89-TR-014, 1989.

[15] J. E. Smith and T. C. Fogarty, "Adaptively Parameterized Evolutionary Systems: Self Adaptive Recombination and Mutation in a Genetic Algorithm," in Voigt, Ebeling, Rechenberg, and Schwefel, Eds., *Parallel Problem Solving From Nature 4*, pp. 441-450, 1996.

[16] B. Sprunt, L. Sha and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *Real-Time Systems*, vol. 1 (1), pp. 27-60, 1989.

[17] 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*, pp. 285-288, 1998.

[18] M. Wall, "GAlib: A C++ Library of Genetic Algorithm Components," Massachusetts Institute of Technology, http://lancet.mit.edu/ga/dist/galibdoc.pdf, August, 1996.

[19] 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.