
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
{briand, labiche, mshousha}@sce.carleton.ca

Abstract

Reactive real-time systems must react to external events within time constraints: Triggered tasks must execute within deadlines. Through performance stress testing, the risks of performance failures in real-time systems are reduced. We develop a methodology for the derivation of test cases that aims at maximizing the chances of critical deadline misses within a system. This testing activity is referred to as performance stress testing. Performance stress testing is based on the system task architecture, where a task is a single unit of work carried out by the system. The method developed is based on genetic algorithms and is augmented with a tool, Real Time Test Tool (RTTT). Case studies performed on the tool show that it may actually help testers identify test cases that are likely to exhibit missed deadlines during testing or, even worse, ones that are certain to lead to missed deadlines, despite schedulability analysis assertions.
1 - Introduction

System testing has been the topic of a myriad of research. With the plethora of real-time applications abounding, testing real-time systems has - in turn - received its own share of investigation. Most testing approaches target system functionality rather than performance. However, Weyuker and Vokolos point out in [1], that a working system more often encounters problems with performance degradation as opposed to system crashes or incorrect system responses. In other words, not enough emphasis is generally placed on performance testing. In hard real-time systems, where stringent deadlines must be met, this poses a serious problem. Examples of hard real-time systems vary from aircraft controllers to nuclear power plant controllers [6]. Because hard real-time systems are often safety critical systems, performance failures are intolerable. Deadlines that are not adhered to can in some applications lead to life-threatening problems. The risk of this occurring can be greatly reduced if higher emphasis is placed on performance testing.

The need for performance testing is deemed even greater when task schedulability is inaccurate. As Section 4.1 reveals, task sets that are thought to always conform to their deadlines might actually wind up missing deadlines. This behavior arises when aperiodic tasks are treated, for schedulability analysis, as equivalent periodic tasks. This generally leads to unrealistic situations since the arrival of aperiodic tasks is often unpredictable. Though it is argued that this assumption simulates the worst-case scenario, it is not always the case. Furthermore, task execution times are mere estimates based, for example, on expected lines of code. Hence, using these estimates in schedulability analysis may not accurately reflect reality. In other words, schedulability analysis may reveal that all timing constraints can be met, yet if these estimates are inaccurate, task deadlines may not be met. Because of inaccuracies in execution time estimates and the simplifying assumptions of schedulability theory, it is then important, once a set of tasks have been shown to be schedulable, to derive test cases to stress the system and verify that tasks cannot miss their deadlines, even under the worst possible circumstances. We cannot simply rely on schedulability theory alone. We refer to this testing activity as performance stress testing. To this aim, we develop a methodology that helps identify
performance scenarios that can lead to performance failures in a system. The method we present combines the use of external aperiodic events (ones that are part of the interface of the software system under test, i.e., triggered by events from users, other software systems or sensors) and internally generated system events (events triggered by external events and hidden to the outside of the software system) with a Genetic Algorithm. Hence, the external aperiodic events are manipulated, searching for a combination of these events that, when coupled with internally generated system events, will potentially inhibit a target task from performing within its specified deadlines. The search for an optimal combination of inputs uses a Genetic Algorithm under constraints (e.g., priorities). The specification of test cases for performance stress testing does not require any system implementation to be available; it can thus be derived during design once task architecture is defined. This can occur just after schedulability analysis, once tasks are deemed schedulable. Performance stress testing can then be planned early. Once the implementation is available and after completion of functional testing, performance stress testing may begin right away.

Our approach is augmented with a tool, Real-Time Test Tool (RTTT). RTTT additionally reports hypothetical percentages of execution estimates that will lead to deadline misses.

Hence, the work we present aims at addressing three sub-problems: Finding a combination of inputs that causes the system to delay task completion to the greatest extent possible, determining accuracy of execution time estimates, and finding a test case through automation. Through performance stress testing, the methodology developed and supported tool reduce the risk of performance failures in real-time systems.

1.1 Problem Description

Real-time system testing can be divided into functional testing and performance testing. Throughout the literature, various approaches for functional testing have been suggested and used. Performance testing, on the other hand, has received very little attention. We address the performance testing problem and do so by decomposing it into a number of sub-problems:

1. Finding a combination of inputs that causes the system to delay task completion to the greatest extent possible
2. Determining accuracy of execution time estimates
3. Finding a test case through automation

Finding a combination of inputs that causes the system to delay task completion to the greatest extent possible: The approach to testing hard real-time systems is to find a combination of inputs that lead to the system’s inability to perform a set of tasks within their specified deadlines. System inputs are in the form of events that are managed at the task level, depicting a single unit of work carried out by the system. It is important to note that system-initiated or internal events also exist. Internal events occur when a stimulus internal to the system is activated in reaction to an external event or task. Internal events can only be monitored by observing the state of the system. User inputted external events and system-initiated or internal events are not differentiated as such. Rather, the different tasks composing each are distinguished by their recurring times into periodic and aperiodic. Periodic tasks occur at constant intervals defined by their periods. Aperiodic tasks arrive at various ranges defined by minimum and maximum interarrival times. Unlike periods, minimum and maximum interarrival times define an interval of time during which these tasks may arrive. Due to their cyclic nature, periodic tasks cannot be controlled as they arrive at determined intervals. Triggering of aperiodic tasks, on the other hand, can be manipulated. Hence, it is aperiodic tasks that are used as system inputs and are used to inhibit a particular task – the target task - from adhering to its real-time constraints by triggering them at various times. These inputs must meet predefined timing constraints. For example, consecutive aperiodic task executions must be greater than or equal to the minimum interarrival time defined for the task. If a maximum interarrival time exists, consecutive task executions must lie within its range.

The importance of real-time performance testing is deemed even more important considering the limitations on schedulability. A group of periodic and aperiodic tasks can be deemed schedulable by the Generalized Completion Time Theorem (Section 2.2.2), yet a combination of inputs can be found where task deadlines are not met. Hence, tasks in real-time systems that were once theoretically thought to be time conformant are not.

Hence, our objective can be formulated as: Given a target task, a set of aperiodic tasks, a set of periodic tasks, and time constraints for each, what is the best way to
schedule these tasks that maximizes the time elapsed between the target’s arrival time and its response?

*Determining accuracy of execution time estimates:* Task execution times are estimates. Using estimates in schedulability analysis will reflect what happens in reality only to a certain extent. For example, when using execution estimates for testing a set of tasks, the worst-case scenario may reveal that all timing constraints can be met. However, if these estimates are inaccurate – that is off by a certain percentage, task deadlines may not be met. A means of determining the measure of precision of these estimates is needed; one that will determine to what degree the estimates cause no deadline misses. In other words, the problem can be reformulated as: what is the inaccuracy percentage of execution estimates that will lead to deadline misses?

*Finding a test case through automation:* Formulating a performance stress test case is not simple. Rather, it involves a number of steps. The target task is chosen by the user as one that poses a higher threat if its time constraints are not met. From there, a sequence of inputs that forces the target task’s executions close to their deadlines is formulated. As Section 4.2 will show, this can be both highly time inefficient as well as tedious when manually applied. The problem becomes even more difficult when a different input sequence strategy must be customized for each individual problem. In other words, a simple strategy does not exist that determines a sequence of inputs that will lead to deadline misses. Furthermore, a number of timing considerations must be considered when determining a scenario of inputs. Input triggering times are monitored such that they always conform to the given minimum and maximum interarrival times. Execution times must also be tracked. Similarly, the different priorities and dependencies of tasks must be tracked to determine the tasks eligible to execute. As task numbers abound, tracking these timing constraints manually becomes rather tedious. Thus arises the need for an automated testing method; one that is capable of determining appropriate input sequences and tracking timing constraints.

The performance stress testing strategy presented here addresses both inaccurate execution time estimates as well as schedulability issues. It consists in finding combinations of inputs representing seeding times for aperiodic tasks, such that a specific task completion time is as close as possible to its deadline. This is not an easy problem to
solve as a system typically has numerous periodic and aperiodic tasks, each with different priorities. However, we present a practical way to automate this that allows 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 to be inaccurate, once the system is implemented.

The remainder of this report is divided as follows: Section 2 provides some background on schedulability theory, processor execution concepts and task scheduling strategies. Readers familiar with these concepts may wish to move ahead to Section 3, which presents a survey of related work. Section 4 presents the theory developed in this work: it describes the reasons behind the use of Genetic Algorithms as a solution to the optimization problem. Section 5 describes how the Genetic Algorithm was tailored. Case studies are then reported in Section 6. We draw conclusions and present ideas for future work in Section 7.

2 – Background

This section familiarizes the reader with concepts that are used throughout this paper. We introduce different schedulability theory (Section 2.2). Readers familiar with these concepts may wish to move on to Section 3.

2.1 Task Scheduling Strategies

On a single processor, or CPU, concurrent tasks must be handled by the operating system’s scheduler. The scheduler maintains a list of tasks that are ready to execute based on different task scheduling algorithms. The task at the head of the list is serviced first, followed by the remaining tasks in the list [5]. In round robin task scheduling, all tasks are placed in the ready list based on a first-come-first-served basis [11]. The task at the head of the list is serviced until its allocated run time elapses, or it blocks, waiting for a resource. If the task has not completed its execution, it is suspended by the kernel and added to the end of the ready list. The head of the ready list is then serviced in the same manner. Round robin scheduling ensures fair allocation of resources for all tasks [5]. However, for real-time systems, this is not a primary concern. In such systems, time critical tasks must be serviced quickly to ensure they meet their deadlines. Real-time systems use priority pre-emption scheduling, where tasks are assigned priorities according to their importance as deemed by the designer or developer, and the ready list
is split into a number of ready queues. Here, a ready queue is maintained for each priority level. When a task becomes ready to execute, it is placed in its priority’s ready queue. The scheduler determines the task that will be serviced by retrieving the head of the highest priority task’s ready queue [4]. This task is serviced until it blocks or until it is pre-empted by a higher priority task [5]. Equal priority tasks are serviced using either round robin scheduling [5], or based on the earliest deadline first where tasks with shorter deadlines are serviced first [4].

Variations on priority pre-emption scheduling exist. In fixed-priority scheduling, tasks are assigned priorities during creation [4]. The assigned priority may be based on the task’s period (as in the Rate Monotonic Algorithm discussed in Section 2.2.1) or heuristically based on the designer or developer’s intuition [9]. Some scheduling techniques allow fixed priorities to become dynamic, or adjustable, to a certain extent. In dynamic priority scheduling, priorities are used to ensure that indefinite blockage of higher priority tasks by dependent lower priority tasks, does not occur. Dependent tasks are ones that share resources, such as shared memory, by means of critical sections [5]. When a task enters a critical section, it has the exclusive right to run without being interrupted by any task that is dependent on it. For example, assume t1 and t2 are two dependent tasks with the former of higher priority. Assume t2 is executing. To ensure that t2 does not hold up t1 for an arbitrarily long time, t2’s priority is adjusted, becoming of higher priority than t1. If dependencies occur between more than two tasks, the priority of the executing task would be increased to the highest priority of all the tasks blocked by it [5].

Because a variety of scheduling techniques abound, the IEEE Computer Society created the Portable Operating System Interface Standard (POSIX), or POSIX 1003 as it is commonly known. POSIX 1003.1, based on the Unix operating system, defines the basic services all operating systems should have and addresses real-time systems in particular in an extension, POSIX 1003.1b. Among the services provided in POSIX 1003.1b is fixed priority pre-emption scheduling [4,5] with round robin scheduling for dealing with equal priority tasks [4]. Priority level numbering is such that higher numbers are indicative of higher priorities. A minimum of at least 32 priority levels must exist for
operating systems to be POSIX compliant. Incidentally, all modern operating systems meet POSIX standards in supporting fixed priority pre-emptive scheduling [4].

2.2 Schedulability Theory

Schedulability theory determines whether a group of tasks, with known CPU utilization times, will meet their deadlines. 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. The underlying scheduling algorithm applied to determine the schedulability of tasks is fixed priority pre-emptive.

It is important to note that all task CPU execution times used for schedulability theory are estimates. These estimates may be obtained by approximating the execution time of the compiled lines that correspond to the original source code [5]. Estimates additionally account for internal events. Internal events can only be monitored by observing the state of the system. Nevertheless, their estimated response times are incorporated into the execution estimates of tasks.

2.2.1 Rate Monotonic Algorithm

The Rate Monotonic Algorithm assigns priorities to periodic tasks based on their periods. Tasks with shorter periods are assigned higher priorities than tasks with longer periods. This ensures that shorter period tasks will not be blocked by longer period tasks. However, not all tasks considered for scheduling are periodic. The Rate Monotonic theory is extended to deal with aperiodic tasks. In particular, [5,6,7,8] show that, for schedulability analysis, aperiodic tasks can be modeled as periodic tasks with periods equivalent to the minimum interarrival time. There are, however, limitations to this as illustrated in Section 4.1.

Sometimes, tasks may be assigned actual priorities that are different from their rate monotonic ones. This alteration of priorities may lead to problems when tasks are dependent. Recall that dependent tasks are tasks that share a resource or critical section [5]. Hence, a task of high priority may not be able to execute because a lower priority task is either using the shared resource or is executing in the critical section. The high priority task is blocked. This situation is dubbed rate monotonic priority inversion. To account for rate monotonic priority inversion, where task blocking by lower priority tasks
may abound, the basic algorithms utilizing the Rate Monotonic Algorithm are further extended [5]. In particular, they are extended to account for four factors:

1. Pre-emption by higher priority tasks with smaller periods
2. Pre-emption by higher priority tasks with longer periods
3. Execution time for the task
4. Blocking time by lower priority tasks (due to rate monotonic priority inversion)

### 2.2.2 Generalized Completion Time Theorem

The Completion Time Theorem (CTT) is one of the basic algorithms that employs the Rate Monotonic Algorithm. Its extension, the Generalized Completion Time Theorem (GCTT), accounts for priority inversion. This is achieved by extending the basic Completion Time Theorem to account for the blocking effect from lower priority tasks as well as pre-emption by higher priority tasks that do not observe Rate Monotonic priorities. The Generalized Completion Time Theorem determines whether a task can complete execution by the end of its period, given pre-emption by higher priority tasks and blocking time by lower priority tasks. The theorem assumes the worst case where all tasks are ready for execution at the start of the task’s period. So, at time zero, all tasks are awaiting execution [5]. It is important to note that the Generalized Completion Time Theorem can only determine whether a group of tasks are schedulable. Hence, if the Generalized Completion Time Theorem check fails, the group of tasks at hand may be schedulable. It does not guarantee that they are not schedulable.

In the mathematical formulation of the Completion Time Theorem, a check is performed to see whether a given task, $\tau_i$, can complete its execution by its first deadline, at worst case, or by the periods of higher priority tasks that lie within its deadline, taking blockage time by lower priority tasks into consideration. The theorem checks tasks in decending priority, with the highest priority task checked first. The mathematical formulation of the Completion Time Theorem is:

A set of $n$ periodic tasks can be scheduled by the Rate Monotonic Algorithm\(^1\) for all task phasing if the following condition is satisfied:

---

\(^1\) The Rate Monotonic Algorithm is extended to deal with priority inversion through the Generalized Completion Time Theorem.
\( \forall i, 1 \leq i \leq n, \min_{(k,l) \in R_i} \left( \sum_{j=1}^{i-1} C_j \frac{1}{LT_j} \left\lfloor \frac{LT_k}{T_j} \right\rfloor \right) + \frac{C_i}{LT_k} + \frac{B_i}{LT_k} \leq 1 \) \hspace{1cm} (1)

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

Hence, for a given task \( i \), \( C_i \) is the execution time of \( i \), \( C_j \) and \( T_j \) are the execution time and period of task \( j \), \( k \) is the task index of \( i \) as well as all higher priority tasks \( (1 \leq k \leq i) \), \( l = 1, \ldots, \left\lfloor \frac{T_i}{T_k} \right\rfloor \) and denotes the scheduling points of task \( k \), and \( B_i \) is the worst-case blocking time for \( i \) [7].

In equation 1, the first term deals with all higher priority tasks. Here, the execution time multiplied by the number of times a higher priority task can execute within task \( i \)'s deadline is summed for each higher priority task. In essence, this identifies the amount of time that will be spent servicing higher priority tasks. The second term adds the execution time of \( i \), thus ensuring that it will indeed meet its deadline. The final term accounts for time \( i \) spends blocked by lower priority tasks.

Let us consider an example with two independent tasks, \( t_1 \) and \( t_2 \) with periods 10, 20 respectively and execution times 1, 2. We know that a task is schedulable if substituting in equation 1 yields a true inequality. Hence, we only need one true inequality to determine whether a task is schedulable or not. For \( t_1 \), \( i=1, k=1, \) and \( l=1, \ldots, \left\lfloor \frac{T_i}{T_k} \right\rfloor = 1 \). Substituting values for the variables in equation 1, \( \frac{1}{10} + 0 \leq 1 \), i.e. \( 1 \leq 10 \). This is a true statement, meaning that \( t_1 \) will be able to meet its deadline. Hence, \( t_1 \) is schedulable. It is important to note that because \( t_1 \) and \( t_2 \) are independent – that is, they do not share resources or critical sections - the blocking time is zero. Now, we check whether \( t_2 \) is also schedulable. For \( t_2 \), \( i=2, k=1, 2 \). Here, we will obtain values for \( l \) twice, one where \( k=1 \) and one where \( k=2 \). \( l_{k=1} = 1, \ldots, \left\lfloor \frac{20}{10} \right\rfloor = 1, 2 \). We first see whether using \( l=1 \) will yield a true inequality. Substituting we obtain \( \frac{1}{10} \left\lfloor \frac{10}{10} \right\rfloor + \frac{2}{10} + 0 \leq 1 \), i.e. \( 1 + 2 \leq 10 \). Again, this is a true statement, meaning that \( t_2 \) will be able to meet its deadline before
first \((l=1)\) deadline of \(t_1\) \((k=1)\). Hence, both \(t_1\) and \(t_2\) are schedulable. If for \(l=1\), \(t_2\) had not yielded a valid inequality, a check for \(l=2\) would be necessary. If that too yielded an invalid inequality, the different \(l\) values for \(k=2\) would need to be checked. If no inequality were valid, \(t_2\) would be unschedulable.

It is interesting to note that equation 1 makes a number of assumptions. The mathematical formulation of the Generalized Completion Time Theorem assumes that the set of tasks are ordered according to increasing priorities. This is evident in the first term, where the bounds of the summation are defined from one to \(i-1\). Furthermore, the mathematical formulation of the Generalized Completion Time Theorem does not explicitly account for equal priority tasks. Should the equal priority tasks be accounted for in the blocking time, or should they be treated as higher priority tasks? This is a drawback to the mathematical formulation. However, the Generalized Completion Time Theorem can account for equal priority tasks through another means: diagrammatically through timing diagrams that show all tasks’ first periods. A timing diagram is a diagram that shows the time-ordered execution sequence of a group of tasks. Figure 1 illustrates an example of a timing diagram. Two tasks are presented, \(t_1\) and \(t_2\). The former is periodic and the latter is aperiodic arriving at time units 20 and 120 respectively. Tasks are illustrated as active throughout the diagram, with shaded portions identifying when the tasks are executing. Timing diagrams are based on UML sequence diagrams, further annotated with time [5].
3 – Related Work

To the best of our knowledge, no existing work addresses the automated derivation of test cases for performance stress testing of real-time, concurrent systems from the perspective of maximizing the chances of missed deadlines. The closest work we are aware of is by Zhang and Cheung [2] who describe a procedure for automating stress test case generation for multimedia systems. The aim of stress testing is twofold: detection of load sensitive faults, as well as ensuring that systems adhere to their performance requirements under heavy loads [3]. A multimedia system consists of a group of servers and clients connected through a network. Stringent timing constraints as well as synchronization constraints are present during the transmission of information from servers to clients and vice versa. The authors identify test cases that can lead to resource saturation, hence adopting a stress testing technique.

Zhang and Cheung model the flow and concurrency control of multimedia systems using Petri nets coupled with temporal constraints. Because the authors model the flow of multimedia systems, the temporal operators they use pertain to multimedia objects as a whole. Thus, temporal operators are defined for the beginning and ending time of a presentation of a media object. Media object duration and starting and ending times with respect to other objects are also defined. For example, given two media objects, VideoA and VideoB, the representation: \( \alpha_{VideoB} = \beta_{VideoA} + 4 \) (where \( \alpha_{VideoB} \) and \( \beta_{VideoA} \) denote the begin time of VideoB and end time of VideoA respectively) is used to express the starting of VideoB four time units after the end of VideoA.

In their model, Zhang and Cheung first identify a reachability graph of the Petri net representing the control flow of multimedia systems. This graph is quite similar to a finite state machine where the states are referred to as markings and the transitions correspond to the transitions in the Petri net. Each marking on the reachability graph is composed of a tuple representing all the places on the Petri net along with the number of tokens held in each. It is important to note that only reachable markings - that is ones that...
can be reached by an execution of the Petri net - are included in the reachability graph. From there, the authors identify test coverage of their graph as a set of sequences that cover all the reachable markings. These sequences, or paths in the reachability graph, are referred to as firing sequences. Firing sequences are composed of a transition and a firing time, represented as a variable. From there, each sequence is formulated into a linear programming problem and solved, outputting the actual firing times that maximize resource utilization.

The authors provide an example illustrating their methodology. They offer the presentation of Figure 2 below modelled as a Petri net.

![Multimedia Petri net example](image)

**Figure 2: Multimedia Petri net example**

A Demo presentation and Full presentation are depicted in the figure. The former is composed of both video and text while the latter contains just video. The presentation time of VideoA is five time units, starting at time 0; the presentation time of VideoB is six time units starting at time unit 9; etc… The reachability graph of Figure 2 is depicted in Figure 3. The markings in the figure denote (Init, Demo, Full, Quit). Hence, the initial marking is given by (1, 0, 0, 0) indicating that there is one token at Init.
From Figure 3, two possible firing sequences are: \{<(\text{selDemo}, t_1), (\text{cxlDemo}, t_2), (\text{selFull}, t_3), (\text{cxlFull}, t_4), (\text{exit}, t_5)>\}, where \(0 < t_1 < t_2 < t_3 < t_4 < t_5\) and \{<(\text{selDemo}, t_1), (\text{cxlDemo}, t_2), (\text{exit}, t_3)>\}, \{(\text{selFull}, t_4), (\text{cxlFull}, t_5), (\text{exit}, t_6)>\}, where \(0 < t_1 < t_2 < t_3 < t_4 < t_5 < t_6\). Then, using linear programming, they determine the schedule of each firing sequence that maximizes resource consumption. However, the authors do not include the detailed description of the linear programming details of this example.

While the technique Zhang and Cheung apply in [2] is intriguing, the target of their technique is stress testing the system. The authors do not model a full real-time system as such. Rather, they model the resources of the real-time system and provide a methodology to maximize its resource utilization. The aim of stress testing is twofold: detection of load sensitive faults, as well as ensuring that systems adhere to their performance requirements under heavy loads [3]. Hence, in stress testing multimedia systems, Zhang and Cheung determine whether different media objects could be active at the same time, using this information to activate the media objects. This type of problem formulation lends itself readily to expression in terms of linear functions and inequalities, hence their use of linear programming. The context and focus of the above work differ from our objective, as our target is reactive, concurrent, real-time systems with soft or hard time constraints. Our objective is to determine how such systems can be stimulated through external events to maximize their chances of deadline misses. Hence, we stress test a system’s performance. In other words, once a set of tasks have been shown to be schedulable, we derive test cases to stress the system and verify that tasks cannot miss their deadlines, even under the worst possible circumstances.
We present our methodology for addressing the three sub problems described in Section 1.1: Finding a combination of inputs that causes the system to delay task completion to the greatest extent possible, determining accuracy of execution time estimates, and finding a test case through automation. In Section 4.1, we present some limitations of schedulability theory that tends to lead to unrealistic situations with regards to the arrival times of aperiodic tasks. Next, in Section 4.2, we illustrate the difficulty in manually stress testing a system through a number of examples. Section 1.1 concludes the section by presenting an overview of techniques for solving optimization problems, with emphasis on our chosen method: Genetic Algorithms.

4.1 Limitations of Schedulability Theory

In [5,6,7,8], the authors state that, for schedulability analysis, aperiodic tasks can be modeled as periodic tasks with periods equivalent to the minimum interarrival time. The authors claim that doing so assumes the worst-case scenario whereby aperiodic tasks constantly arrive at their defined minimum interarrival times. Hence, any lower priority tasks will be pre-empted 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. Let us consider, for example, the following tasks (t1 and t3 are periodic, t2 is aperiodic). The dependency between tasks t1 and t3 is in the form of a shared resource. Thus, one task must fully complete its execution before the other task is allowed to run.

- t1: period = 3, priority = 32, execution time = 1, dependent on t3 for 2 time units
- t2: minimum interarrival time = 8, deadline for execution = 8, priority = 31, execution = 3
- t3: period = 9, priority = 30, execution time = 2, dependent on t1 for 1 time unit

Use of the Generalized Completion Time Theorem (Section 2.2.2) to determine the schedulability of tasks, while modelling the aperiodic task as a periodic task with
period equivalent to the minimum interarrival time reveals that the set of tasks are schedulable. The Generalized Completion Time Theorem further reveals when each task will meet its first deadline with respect to the other tasks: t1 will meet its first deadline before any other task executes, t2 will meet its deadline before the second deadline of t1, and t3 will meet its deadline before the first deadline of t2. The complete proof of the theorem for the example is provided in [32].

The schedulability of these tasks can further be illustrated using a timing diagram as shown in Figure 4. The timing diagrams presented here have slightly different notation than those described in [5]. Task arrival times are indicated on the right hand side of the diagram. Different portions of the same task execution are linked by a dotted line, indicating that all portions belong to the same task execution.

![Timing diagram with aperiodic task modelled as periodic](image)

Figure 4: Timing diagram with aperiodic task modelled as periodic

In the figure, 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 pre-empted by the higher priority task t1 at time unit 3, but it is allowed to complete execution after t1 has completed. Task t2’s first execution ends at time unit 5, long before its deadline at time unit 8. At this point, t3 is now ready to execute and does so. Although the higher priority task t1 arrives at time unit 6 while t3 is executing, it is dependent on t3. Because t1 is
accessing a shared resource, t3 cannot execute before t1 has completed its execution, releasing the shared resource. Hence, t3 is not pre-empted. 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.

However, by shifting the arrival time of the aperiodic task slightly\(^2\), setting its arrival times at time units 2, 11 and 19, t1 – because of its dependence on t3 – is no longer capable of meeting neither its second deadline nor its fifth deadline at time units 6 and 15 respectively as illustrated in Figure 5. It is important to note that these new arrival times still comply with the minimum interarrival time of t2. 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 pre-empts 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. Thus, the use of the Generalized Completion Time Theorem (GCTT) has two implications. One implication is that the first occurrence of events triggering aperiodic tasks happens at the same time as the first occurrence of events triggering periodic tasks: at the very beginning of the observation period during which task completion times are monitored and analyzed. The second side effect is that the interarrival time between two consecutive arrivals of an event triggering an aperiodic task is constant. These two side effects can lead to unrealistic situations since the arrival of events triggering aperiodic tasks is often unpredictable.

\(^2\) The arrival times of the aperiodic task were obtained by running our prototype tool with the three tasks as input (Section 4.1). The input file is included in [32].
Figure 5: Timing diagram with aperiodic task not modelled as periodic

The implications of this are critical. According to the Generalized Completion Theorem and modelling aperiodic tasks as periodic ones, a set of tasks can be deemed schedulable when in reality they are not. Hence, on an application level, for a real-time system, a particular sequence of events of a real-time system can theoretically be considered “safe”, in that all tasks meet their deadlines, when in reality, the situation is far from “safe”. The testing aid we present can safeguard against the occurrence of these situations. Even if a set of tasks is deemed schedulable by the Generalized Completion Time Theorem, our methodology and associated application will determine whether this is truly the case, outputting seeding times that represent the worst solution found which causes tasks to miss their deadlines.

Furthermore, schedulability literature does not address the problem of execution estimates. Because execution times are estimates, possible deadline misses may be hidden. In other words, given execution estimates, a group of tasks may be deemed schedulable through schedulability analysis. However, if the given estimates are inaccurate by a small percentage, deadline misses occur. For example, consider the following independent tasks (t1 and t3 are periodic, t2 is aperiodic):

- t1: period = 64, priority = 32, execution time = 50
• t2: minimum interarrival time = 60, deadline for execution = 60, priority = 31, execution = 5

• t3: period = 62, priority = 30, execution time = 5

The timing diagram for this set of tasks is presented in Figure 6.

Figure 6: Timing diagram illustrating estimates

All tasks successfully complete their executions before their deadlines. Yet, the execution times defined for each of the tasks are mere estimates based on the approximation of compiled lines of code (Section 2.2). If each execution time estimate is off by just 2%, deadlines are missed, as shown in Figure 7.
In Figure 7, because the time estimate for t1 is off by 2%, t1 completes its execution by time unit 51 instead of time 50 as shown in Figure 6. Similarly, t2 begins its execution at time unit 51 and ends at time unit 56.1. Task t3 begins at 56.1 but is pre-empted by the higher priority task t2 at time unit 60. By time unit 62 - t3’s deadline - t2 still has not completed its execution. Task t2 is itself pre-empted by t1 at time unit 64. The diagram does not show when either t2’s second execution completes, or t3’s first execution. However, t3 has clearly missed its first deadline because of the 2% estimate error.
4.2 No Simple Strategy

Performing manual performance stress testing on a real-time system can be rather difficult. In performance stress testing, the system is tested by choosing seeding times for aperiodic tasks that are most likely to result in deadline misses. We depict this difficulty through a number of examples illustrated through timing diagrams in Sections 4.2.1 and 4.2.2. Each of these sections first illustrates a benchmark for the example, followed by the application of the GCTT, and finally a number of manual techniques for testing. The different techniques are then compared. A summary appears in Section 4.2.3. The application of the GCTT and its extension is used as a basis of comparison to assess whether the other manual techniques have detected a more stressful test case.

In the following examples, all tasks are independent. Tasks are periodic, unless marked with (*) in which case they represent aperiodic tasks. Target tasks are the tasks whose deadlines are examined. These are identified with a (T). All tasks are defined with CPU execution times (C), periods (T) and priorities (P). For priorities, the higher the number, the higher the priority. Periods of aperiodic tasks designate their minimum interarrival times.

4.2.1 Example 1

The example presented here follows through from a benchmark for comparison, application of the GCTT, and finally a number of manual performance testing strategies. The manual strategies are ones that a tester would probably apply. These approaches set the aperiodic task’s arrival times to: the arrival times of the target task and the arrival times of the highest priority task. The three tasks comprising this example are:

\* t1: C=1 T=3 P=32

\[ t2: C=1 T=4 P=31 \]

(T) \[ t3: C=1 T=5 P=30 \]

---

3 Based on an example that appears in [10]
A: Benchmark

This subsection serves as a benchmark for comparison with the remainder of this example. It demonstrates how the periodic tasks would execute without the triggering of the aperiodic task, as illustrated in Figure 8.

![Figure 8: Benchmark for Example 1](image)

B: GCTT

In [7], Sha and Goodenough state that the Generalized Completion Time Theorem assumes the worst-case scenario of task initiation whereby all tasks are triggered at the same time: time unit 0. The theorem is extended to account for aperiodic tasks, treating them as periodic for the sake of schedulability analysis. We use the theorem and its extension to try to delay the target task’s executions by identifying triggering times for the aperiodic task as shown in Figure 9. The timing diagram of the figure is extended to show task arrival times on the right of the diagram.

![Figure 9: Application of the GCTT](image)
In Figure 9, the first execution of t3 can only start at time unit 2, ending at time unit 3. In comparison to Figure 8, this is a shift of one time unit closer to its deadline at time unit 5. The second execution of t3 is not altered by the introduction of t1; it begins execution at time unit 5. Hence, t3’s first execution ends two time units away from its deadline while the second execution ends four time units away. By using the Generalized Completion Time Theorem and its extension to determine the arrival times of t1, the likelihood of t3 not being able to meet its deadline is increased. Execution times are estimates. Hence, if the estimate of t3’s execution is off by a large enough percentage (100% in this case because execution estimates are small), t3 might actually miss its deadline.

C: Triggering with Target Task

Because aperiodic tasks are triggered in a random manner, they can arrive at times that are different from those defined by Figure 9. If aperiodic task t1 arrives at precisely the same time as the target task, t3, the target task will be delayed by the execution time of the aperiodic task. Furthermore, if any task of higher priority than the target task arrives, it too will delay the target by its execution times. Figure 10 illustrates this strategy.

![Figure 10: Triggering with target task](image)

In Figure 10, t1 is triggered with t3. The first execution of t3 is not different from that of Figure 9 and t3 ends its first execution at time 3. In its second execution, however, t3 cannot start executing before time unit 6, finishing at time unit 7. Hence, once t3’s second execution is complete, it is closer to its deadline than that of Figure 8 and Figure 9. T3’s second execution is only three units away from its deadline. Therefore, triggering
the aperiodic task with the target task can - in some cases - lead to closer deadline misses than when converting aperiodic tasks to periodic ones.

**D: Triggering with Highest Priority Task**

In this approach, the aperiodic task is triggered at the same time as the highest priority task is triggered, so long as the minimum interarrival time constraint is met. The aim is to delay the target task’s completion for each of its executions as long as possible whereby both the aperiodic task and any tasks of greater priority than the target will delay the execution of the target. In the example, the aperiodic task, t1, is the highest priority task. Because we are searching for seeding times for the aperiodic task, triggering occurs with the next highest priority task, t2, instead, as shown in Figure 11.

![Figure 11: Triggering with highest priority task](image)

Like the previous approach, the target task’s executions end at time units 3 and 7 respectively. Again, this approach causes the target task’s executions to be delayed more than the approach using the Generalized Completion Time Theorem.

### 4.2.2 Example 2

The example presented here proceeds like the example from the previous section. A benchmark for comparison is first presented followed by an application of the GCTT, and finally a number of manual performance testing strategies. The manual strategies set the aperiodic task’s arrival times to: the arrival times of the target task, the arrival times of the highest priority task and one time unit before the end of the execution of the target task. The tasks in this example are\(^4\):

---

\(^4\) Based on an example that appears in [5]
t1: C=20, T=100, P=32
* t2: C=30, T=150, P=31
(T) t3: C=90, T=200, P=30

**A: Benchmark**

Figure 12 shows the set of periodic tasks and their executions. Again, this serves as a benchmark for comparison with other sections of this example.

![Figure 12: Benchmark for Example 2](image)

**B: GCTT**

Applying the worst-case scenario of the Generalized Completion Time Theorem and its extension yields the timing diagram of Figure 13.
In Figure 13, t2 is triggered at its minimum interarrival time: time unit 0 and 150. The presence of t2 causes the target task, t3 to delay its execution start as t2 is of higher priority than t3. Consequently, t3 now completes its execution at time unit 190, just 10 time units from its deadline when without the presence of t2 as seen in Figure 12, t3 would have completed execution much earlier at time unit 130.

**C: Triggering With Target Task**

Triggering t2 at the same time as t3 yields the timing diagram of Figure 14.
The approach applied in Figure 14 for this example guarantees that the target task’s execution meets its deadline earlier than the approach applied in Figure 13. Here, t3 completes its execution at time unit 160, 30 time units before its respective completion in Figure 13. Recall that in example one, this same approach yielded a different outcome; one where task completion time was further delayed. Consequently, this approach - triggering aperiodic tasks at the same time as the target task - is not always successful in providing an adequate solution.

**D: Triggering With Highest Priority Task**

Triggering task t2 with t1 yields the timing diagram of Figure 14. The result is the same as the previous approach. Task t2 is first triggered at time 0. It cannot be triggered at time unit 100 because its minimum interarrival time is greater (150). It can only be triggered at time unit 200 when the timing constraint is met. This allows t3 to fully complete its execution by time unit 160, much earlier than execution end time 190 in the Generalized Completion Time Theorem approach.
E: Triggering One Time Unit Before End of Target Task Execution

If the aperiodic task were triggered at one time unit before the end of the execution of the target task, this could delay the target task’s execution, especially if higher priority tasks cause further pre-emption. This approach was applied to the example yielding the timing diagram of Figure 15.

![Timing Diagram](image)

**Figure 15: Triggering before end of target task**

In Figure 15, t3’s execution would have completed at time unit 130. Hence, following the approach, t2 is triggered at time unit 129 as depicted. Task t2 completes its execution without pre-emption, allowing t3 to complete its execution at time unit 160. Once again, as with the Generalized Completion Time Theorem approach, t3’s execution is not delayed beyond time 160. Again, it is important to note that this approach may not always be applicable. In example one, the target task’s execution was precisely one time unit. Because execution times are integers, the approach presented here would not work.

4.2.3 Summary

The previous two examples yielded different results. In example one, triggering the aperiodic task at the same time as the target task and highest priority task caused the target to be delayed more than triggering the aperiodic task as a periodic one. Hence, the
former two approaches are better in causing deadline misses in the target. In example two, on the other hand, triggering the aperiodic task at periodic times caused more delay in the target than triggering at the same time as the target task and highest priority task. It also yielded more delay than triggering one time unit before the end of the target’s execution. Hence, in example two, the Generalized Completion Time Theorem approach was better in causing deadline misses in the target. It can thus be observed that the Generalized Completion Time Theorem and its extension will not always yield the worst execution ending times in the target task. In the different scenarios of examples one and two, the Generalized Completion Time Theorem approach was in some cases better than other approaches and in some cases worse. Each approach is further seen to yield good results at times and bad results at others. A single approach does not exist that can determine the input sequences that will lead to the worst-case scenario. In other words, we need to answer the following question: what sequence of inputs will cause the greatest delays in the executions of the target task? This is formulated as an optimization problem, one where the object is to find the best of all possible solutions [23]. Several methods exist that aid in the solution of optimization problems. We next investigate the applicability of genetic algorithms in achieving a solution to our optimization problem.

5 -Tailored GA

A Genetic Algorithm is used to solve the optimization problem of finding seeding times for aperiodic tasks that increase the likelihood of deadline misses in a target task. This section addresses the different components of the GA, tailoring each to the problem. We first present the notation used that we adapt in Section 5.1. In Section 5.2, we define the various components of the GA from chromosome representation, development of the initialization procedure, mutation operators, crossover operators and fitness function.

5.1 Notation

Throughout the remainder of this paper, various notation and terminology are employed. We define these here, indicating their respective meanings.

We define a task to be a single unit of work carried out by the system. All tasks, whether aperiodic, $A_i$, or periodic, $P_i$ have CPU execution estimates $C_i$. They also have priorities $p_i$. In the testing time interval, $T$, each task, i, has a defined maximum number
of executions, \( k_i \), each denoted with a \( j \) index. Hence, the \( j^{th} \) execution of the \( i^{th} \) aperiodic task would be denoted \( A_{i,j} \). Similarly, the \( j^{th} \) execution of the \( i^{th} \) periodic task would be denoted \( P_{i,j} \). Task executions represent different runs of the same task. One execution is defined for each arrival time of the task.

For every task execution, an additional set of variables is defined. Each task execution has a deadline, \( d_{i,j} \), arrival time, \( a_{i,j} \), execution start time, \( s_{i,j} \), and execution end time, \( e_{i,j} \). Deadlines define the time limit by which the task execution must be complete. The execution’s arrival time indicates the time the event triggering the task arrives, while execution start times define the execution’s start time as determined by the scheduler. End times determine the time unit at which executions complete. The values of \( a_{j} \) and \( s_{j} \) do not always coincide. In fact, they only coincide if the corresponding task is the highest priority available task ready to run. For example, consider the task set of \( t_1 \) (period = 50) and \( t_2 \) (period = 40) in Figure 20, with a testing interval of 110 time units. Both events triggering the tasks are assumed to initially arrive at time unit 0. Task \( t_1 \) has three executions: \( a_{1,1} = 0 \), \( a_{1,2} = 50 \) and \( a_{1,3} = 100 \). For \( t_2 \), \( a_{2,1} = 5 \), \( a_{2,2} = 40 \) and \( a_{2,3} = 80 \). It is important to note that in \( P_{2,2} \), \( t_2 \) is pre-empted by \( t_1 \) at time unit 50 and resumes execution at time 55. This resumption, as indicated in the figure, is part of the same execution. It is not considered a different execution.

![Figure 20: Illustration of task execution definition](image)
Periodic tasks additionally have periods, $r_i$. It is important to note that for periodic tasks, $d_{i,j}$ and $a_{i,j}$ are both multiples of the task’s period. Referring to Figure 20, the first execution of $t_1$ with $a_{1,1} = 0$ has its deadline at time unit 50, which marks the period of $t_1$, $a_{1,2} = 50$ and $d_{1,2} = 100$, etc… This relationship between $a_{i,j}$ and $d_{i,j}$ can be expressed with equations 2 and 3 below.

For $P_i$, the following holds:

$$a_{i,j} = (j-1)r_i \quad (2)$$

$$d_{i,j} = jr_i \quad (3)$$

Aperiodic tasks, on the other hand, have minimum interarrival times $\min_i$. They may additionally have maximum interarrival times $\max_i^5$. The values of $\min_i$ and $\max_i$ define the minimum (maximum, respectively) time interval between two consecutive execution arrivals of aperiodic task $i$. Aperiodic tasks also define deadlines. However, unlike their periodic counterparts, when the aperiodic task is defined, the value given for the deadline is absolute. In other words, a deadline value of $x$ defined for $A_i$ indicates that for each task execution, $j$, the deadline is $x$ time units after $a_{i,j}$. The value of the absolute deadline of $A_i$ is denoted as $d_i$. The value of $d_i$ is then used to define the relative deadline value for each execution. For example, consider an aperiodic task with $d_i = 250$. For each execution, $d_{i,j} = a_{i,j} + 250$. Hence, the relationship of equation 4 holds for aperiodic tasks.

For $A_i$

$$d_{i,j} = a_{i,j} + d_i \quad (4)$$

Furthermore, a relationship exists between deadlines and interarrival times, as defined by the inequality in equation 5.

$$d_i \leq \min_i \leq \max_i \quad (5)$$

Target tasks are depicted as $t$. In summary, we present Table 1 with the various terms and their meanings.

---

5 An example where a maximum interarrival time would be specified is a heart pacer. A heart pacer is a machine that is directly connected to a heart that has been known to have beating irregularity. The heart pacer is inactive as long as it detects pulses from the heart. If a pulse is not sensed, the pacer waits for a maximum specified time before it initiates an electric pulse that is sent to the heart. Hence, pulses arrive at the maximum interval defined by the pacer.
### Notation and Reference

<table>
<thead>
<tr>
<th>Notation</th>
<th>Reference</th>
<th>Notation</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>Target task</td>
<td>( min_i )</td>
<td>Minimum interarrival time</td>
</tr>
<tr>
<td>( A_i )</td>
<td>Aperiodic task</td>
<td>( max_i )</td>
<td>Maximum interarrival time</td>
</tr>
<tr>
<td>( P_i )</td>
<td>Periodic task</td>
<td>( d_i )</td>
<td>Absolute deadline</td>
</tr>
<tr>
<td>( T )</td>
<td>Testing time interval</td>
<td></td>
<td>For Periodic Tasks</td>
</tr>
<tr>
<td>( C_i )</td>
<td>CPU execution estimate</td>
<td></td>
<td>Period</td>
</tr>
<tr>
<td>( p_i )</td>
<td>Task priority</td>
<td></td>
<td>For Task Execution ( j ) of task ( i )</td>
</tr>
<tr>
<td>( k_i )</td>
<td>Maximum number of executions</td>
<td>( d_{i,j} )</td>
<td>Execution deadline for task ( i )</td>
</tr>
<tr>
<td>( A_{i,j} )</td>
<td>( j^{th} ) execution of ( i^{th} ) aperiodic task</td>
<td>( a_{i,j} )</td>
<td>Arrival time for task ( i )</td>
</tr>
<tr>
<td>( P_{i,j} )</td>
<td>( j^{th} ) execution of ( i^{th} ) periodic task</td>
<td>( s_{i,j} )</td>
<td>Execution start time for task ( i )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( e_{i,j} )</td>
<td>Execution end time for task ( i )</td>
</tr>
</tbody>
</table>

**Table 1: Summary of terms**

### 5.2 GA Components

In this section, we describe the various constituting components of our GA. We begin by specifying the population size. Determining the population size of the GA is challenging. A small population size will cause the GA to quickly converge on a local minimum because it insufficiently samples the parameter space. A large population, on
the other hand, causes the GA to run longer in search for an optimal solution. Haupt and Haupt in [18] list a variety of work that suggests an adequate population size. The authors in [18] reveal that the work of De Jong suggests a population size ranging from 50 to 100 chromosomes. Haupt and Haupt further state that Grefenstette altered the range between 30 and 80, while Schaffer and his colleagues suggest a lower population size: between 20 and 30 [18]. The population size we apply is 80. This number is consistent with most of these findings.

As our optimization problem does not entail a defined optimum solution, we adopt number of generations as our termination criterion. Once 500 generations have been generated, the GA halts yielding the best score found.

It is important to note that a number of constraints confine the search space of our optimization. These constraints must be adhered to in all instances of the solution. If these constraints are not met, the resulting solution will be considered nullified.

**Constraint 1:** Two consecutive arrival times, $a_{i,j}$ and $a_{i,j+1}$, of an aperiodic task $A_i$ must have a difference of at least $min_i$.

**Constraint 2:** Two consecutive arrival times, $a_{i,j}$ and $a_{i,j+1}$, of an aperiodic task $A_i$ must have at most a difference of $max_i$, if it exists.

Furthermore, it is important to note that the group of tasks under test, both periodic and aperiodic, may or may not be schedulable under the Generalized Completion Time Theorem and its extension (Section 2.2.2). Recall from Section 4.1 that tasks deemed schedulable by the extension of the GCTT may not be so. If they are schedulable by the GCTT, RTTT will attempt to confirm whether this is really the case and will identify stress test cases. If, on the other hand, they cannot be deemed schedulable by the GCTT and its extension, we do not know whether the tasks are schedulable. We use our methodology to investigate whether we can find a test case where deadline misses occur. If we cannot find such a test case, this does not guarantee that none exist. However, one can still feel more confident that such a case is unlikely.

### 5.2.1 Chromosome

Chromosomes define a group of values to be optimized [18]. In our application, the values to be optimized, or the genes of the chromosome, are the arrival times of all aperiodic tasks. Thus, we need to encode both task number and arrival times.
A gene can visually be depicted as a pair of integer values \((A_i, a_j)\). Together, the pair represents an execution of an aperiodic task. If an execution of an aperiodic task does not occur, i.e. is not triggered, the size of the chromosome will fluctuate. Yet, because chromosomes are homologous, we want to maintain a constant chromosome size [20]. This will facilitate the crossover operation presented in Section 5.2.4.1. Thus, we need a special value to depict a non-existent arrival time. In other words, we need a value to indicate that a task execution is not triggered. Because, by definition, all arrival times are greater than or equal to zero, we allocate \(-1\) as this special value. Hence, \((A_i, -1)\) represents a task execution that is not triggered.

To ensure that the optimization constraints of Section 5.2 are met, we introduce two more constraints on chromosomes.

**Constraint 3:** In a chromosome, all genes corresponding to the same task are grouped together.

**Constraint 4:** All genes within each group of genes comprising a task in a chromosome are ordered in increasing order according to \(a_{ij}\).

Constraint 3 will ease the implementation of the crossover and mutation operators (Sections 5.2.4.1 and 5.2.4.2). Likewise, constraint 4 will facilitate the implementation of the operators. Furthermore, by ordering each task’s executions according to increasing arrival times, it will be easier to ensure that the minimum and maximum time constraints of constraints 1 and 2 are met. It is important to note that based on constraint 4, because task executions that are not triggered are assigned the value of \(-1\), all these executions will appear at the beginning of the corresponding task’s executions. Hence, all task executions of the form \((A_i, -1)\) appear at the beginning of the executions of task \(A_i\), followed by increasing arrival times for the following executions.

Consider, for example, a set composed of two aperiodic tasks \(t_1\) \((\text{min}_t = 10, C_{t_1} = 3, p_{t_1} = 32)\) and \(t_2\) \((\text{min}_t = 11, C_{t_2} = 5, p_{t_2} = 31)\). In a time interval of 30, the following is a valid chromosome: \((t_1, -1) (t_1, 19) (t_1, 29) (t_2, -1) (t_2, -1) (t_2, 10)\). The chromosome meets all constraints.

The metamodel describing the relationships between the various concepts is presented in Figure 21.
Chromosome is composed of a sequence of Gene ordered in the fashion described in Section 0. The initialization, crossover and mutation operators are all defined in Chromosome, as well as the objective function, dubbed Evaluate. The Chromosome object uses Task, which represents a task in the tested application. Task is further specialized into AperiodicTask and PeriodicTask. Although only aperiodic tasks are used in the chromosome representation, information pertaining to periodic tasks is needed in the scheduling process. Alternative representations of the chromosome are presented in [32].

Figure 21: Metamodel of relationships
5.2.2 Chromosome Length

As the chromosome holds the arrival times of all aperiodic tasks, the chromosome’s size is the total number of executions of all aperiodic tasks. The maximum number of executions of one aperiodic task, i, can be expressed as

\[ k = \left\lfloor \frac{T}{\min_i} \right\rfloor \]

where \( T \) is the time interval during which testing is performed.

Equation 6 defines the maximum number of executions for i, given that all the executions arrive periodically at the minimum interarrival time (i.e. \( a_{ij} = (j-1) \min_i \)). For example, consider the aperiodic task set\(^6\) of Figure 22 where \( T = 200 \), with tasks t1 (\( \min_{t1} = 100 \), \( C_{t1} = 20 \), \( p_{t1} = 32 \)), t2 (\( \min_{t2} = 150 \), \( C_{t2} = 30 \), \( p_{t2} = 31 \)) and t3 (\( \min_{t3} = 200 \), \( C_{t3} = 90 \), \( p_{t3} = 30 \)). Because t1 has a minimum interarrival time of 100 time units, it can execute a maximum of two times in the given time interval of 200. Likewise, t2 can execute a maximum of two times and t3 can execute a maximum of only once within the allocated time interval.

![Figure 22: Total number of executions example](image)

The maximum number of times all aperiodic tasks can execute is the sum of equation 6:

---

\(^6\) Adapted from an example in [5]
\[ l = \sum_{i=1}^{n} \left\lfloor \frac{T}{\text{min}\alpha_i} \right\rfloor \quad (7) \]

where

- \( n \) is the number of aperiodic tasks and
- \( T \) is the time interval defined for testing.

Equation 7 represents the length of the chromosome.

Using equation 7 in the example of Figure 22, the length of the chromosome is:
\[ \left\lfloor \frac{200}{100} \right\rfloor + \left\lfloor \frac{200}{150} \right\rfloor + \left\lfloor \frac{200}{200} \right\rfloor = 2 + 2 + 1 = 5. \]
It is important to note that this is a maximum length that cannot be exceeded. If the aperiodic tasks do not execute at their minimum interarrival times, some executions will not occur. As discussed in Section 0, these will have arrival time values equal to –1. We considered another way to define the length of the chromosome. This alternate is presented in [32].

As a guideline, choice of the ideal time interval would be one based on the repetition of task execution patterns as visualized on a timing diagram. For example, consider the timing diagram of Figure 23 where both tasks are periodic; \( t_1 \), with the higher priority, has a period of four time units and \( t_2 \) has a period of eight time units. The period of repetition of executions here is eight. Hence, every eight time units, the pattern of execution of tasks \( t_1 \) and \( t_2 \) repeats. After the first eight time units, we know the tasks will repeat executions in exactly the same manner as before. This repetition period is defined by the Least Common Multiple\(^7\) (LCM) of the periods of all tasks.

\[^7\text{The Least Common Multiple (LCM) of two numbers } a \text{ and } b \text{ is the smallest number } m \text{ for which there exists positive integers } n_a \text{ and } n_b \text{ such that } n_a \ a = n_b \ b = m.\]
It is important to note that using the LCM to define the time interval period has its drawback: it assumes that the task set is strictly periodic. Aperiodic tasks, due to their sporadic nature, may alter the repetition period. Aperiodic tasks may even eliminate a repetition pattern altogether.

5.2.3 Chromosome Initialization

The chromosome initialization process ensures that all the constraints of Sections and 5.2 and 0 are met. For each $A_i$ within $T$, $k_i$ is calculated using equation 6, and $k_i$ new genes are then created for each task, $i$, with the first number of the constituting pair set to the task number. The value of $a_{i,j}$ is then 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 $T$. The range is determined as follows: $[a_{i,j-1} + min_i, a_{i,j-1} + max_i]$. If there is no previous gene, or the previous gene is not triggered (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 set to -1 and that execution is moved to the head of the array of $k_i$ genes, hence maintaining the increasing order of arrival times constraint. For example, consider the initialization of an aperiodic task $t1$ ($min_{t1} = 10$, $C_{t1} = 2$, $\rho_{t1} = 32$) with $T = 30$. The value of $k_i$ (using equation 6) is 3.
Thus, three empty genes are created and initialized with the task number: (1, ) (1, ) (1, ). Looping on each of the genes, we then initialize $a_{i,j}$. Because there is no previous gene, the value of $a_{1,1}$ is randomly chosen from the range $[0, \text{max}_1]$ (where because $\text{max}_1$ is not specified, the range is $[0, 30]$). Hence producing: (1, 15) (1, ) (1, ). Similarly, for the second gene, the value of $a_{1,2}$ is randomly chosen from the range $[25, 45]$ ($[a_{1,1} + \text{min}_\alpha_1, a_{1,1} + \text{max}_\alpha_1]$). 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 $[37, 57]$. Let us assume the value chosen from the range is 48. This value 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 head of the array yielding (1, -1) (1, 15) (1, 27). All constraints are upheld.

5.2.4 Operators

Operators – crossover (Section 5.2.4.1) and mutation (Section 5.2.4.2) - are the ways GAs explore a solution space [18]. 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 in a chromosome, backtracking is performed to first invalidate the operation, then to reapply the operation to another chromosome. Backtracking, however, has its drawbacks: it is deemed expensive as well as time consuming. 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. If the application of the operator does not alter the chromosome or produces a chromosome that violates a constraint, we do not commit the changes and search for a different chromosome, or gene within the chromosome – depending on the operator – and reapply the operation. Because of the drawbacks of backtracking, we aim at minimizing its use.

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 [21]. Furthermore, operators should be formulated such that they explore the whole solution space.
5.2.4.1 Crossover Operator

Crossover operators aim at passing on desirable traits or genes from generation to generation [18]. Varieties of crossover operators exist, such as sexual, asexual and multiparent [14]. 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 is to divide both parent chromosomes into two or more fragments and create two new children by mixing the fragments [18]. Pawlowsky dubs this n-point crossover. 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 [22].

In our application, the number defining \( n \) is determined by the number of aperiodic tasks being scheduled. Hence, if the number of aperiodic tasks is \( n \), the resulting crossover operator (using Pawlosky’s terminology) is \((n-1)\) point. The actual division points of the parents depend on \( k_i \) for each of the \( n \) tasks. The first point of division occurs after \( k_{i1} \); the second point of division after \( k_{i1} + k_{i2} \) from the first gene; the \( n-1 \) point of division after \( k_{i1} + k_{i2} + \ldots + k_{iz-1} \) from the first gene. In this manner, all executions of a task are left intact.

In our application, the mixing of the fragments is additionally subject to a number of constraints (constraint 3 in Section 0, and constraints 1 and 2 in Section 5.2). Hence, to avoid mixing task executions, all executions of the same task must belong to the same fragment. In other words, the two parent chromosomes must be divided into fragments based on the number of tasks in the chromosome. This is presented as one crossover operator, \( nPointCrossover \), in our methodology. To further introduce an element of randomness, we alternate the genes of the parents with a 50% probability, hence implementing a second crossover operator, \( nPointProbCrossover \). In \( nPointCrossover \),
the resulting children have fragments that alternate between the parents. Effectively, even sets of task executions are inherited from parents to children. In \textit{nPointProbCrossover}, the same alternation pattern occurs as \textit{nPointCrossover}. However, instead of always inheriting a fragment from a parent, children inherit fragments from parents with a probability of 50%. This can be visualized as a coin flip. When alternating the fragments of each parent, a coin is flipped. Every time the coin lands on heads, the fragment is inherited by the child. Otherwise, the fragment is not inherited. It is important to note that, for both crossover versions, if task executions are not maintained within the same fragment in the parents, allowing partial crossover in the children, constraints 1 and 2 (Section 5.2) will often be violated, necessitating backtracking.

Let us consider an example with three tasks\footnote{Based on an example in [5]} $t_1$ ($min_{t1} = 100$, $C_{t1} = 20$, $p_{t1} = 32$), $t_2$ ($min_{t2} = 150$, $C_{t2} = 30$, $p_{t2} = 31$) and $t_3$ ($min_{t3} = 200$, $C_{t3} = 90$, $p_{t3} = 30$). Because the number of aperiodic tasks is three, two-point crossover is applied dividing the parents as shown in Figure 24.

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node[above] at (0,0) {Parent 1};
\node[below] at (0,-1) {Child 1};
\node[above] at (1,0) {Parent 2};
\node[below] at (1,-1) {Child 2};
\node[above] at (2,0) {Child 1};
\node[below] at (2,-1) {Child 2};
\node[above] at (3,0) {nPointCrossover};
\node[above] at (3,-1) {nPointProbCrossover};
\node[above] at (4,0) {nPointCrossover};
\node[above] at (4,-1) {nPointProbCrossover};
\draw (0,0) -- (1,0) -- (2,0) -- (3,0) -- (4,0);
\draw (0,-1) -- (1,-1) -- (2,-1) -- (3,-1) -- (4,-1);
\draw (0,0) -- (0,-1);
\draw (1,0) -- (1,-1);
\draw (2,0) -- (2,-1);
\draw (3,0) -- (3,-1);
\draw (4,0) -- (4,-1);
\end{tikzpicture}
\caption{Illustration of crossover operators}
\end{figure}

In \textit{nPointCrossover}, the fragments of Parent 1 and Parent 2 are alternately interchanged. Using the same example for \textit{nPointProbCrossover}, one possible outcome appears in Figure 24. In the figure, the coin flips are assumed to land on heads, tails, then tails for the three successive fragments for both children.
The advantages of *nPointProbCrossover* are twofold. It introduces further randomness to the crossover operation. By doing so, it allows further exploration of the solution space. Furthermore, *nPointProbCrossover* is a generalized version of *nPointCrossover*; if the coin flip for each fragment alternates between tails and heads (in that order), we obtain *nPointCrossover*. However, *nPointProbCrossover* has its disadvantages. If the result of all coin flips in a given operation is always tails or always heads, the resulting children are replicas of the parents, with no alteration occurring. Never is this the case with *nPointCrossover*; resulting children are always genetically distinct from their parents. Comparison of the two crossover techniques revealed that *nPointProbCrossover* converged somewhat faster than *nPointCrossover*.

Crossover rates are critical. If the crossover rates are too high, desirable genes will not be able to accumulate within a single chromosome whereas if the rates are too low, the search space will not be fully explored [18]. Haupt and Haupt [18] list several findings of crossover rates. The authors state that, from his work, De Jong concluded that a desirable crossover rate should be about 60%. Grefenstette built on De Jong’s work and found that crossover rates should range between 45% and 95%. Later, working in a group Schaffer and his colleagues reported that ideal crossover rates should range between 75% and 95% [18]. Consistent with the findings of De Jong and Grefenstette, we apply a crossover rate of 70%.

### 5.2.4.2 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 chosen for mutation, the gene is mutated, and the resulting chromosome is evaluated for its new fitness. We define a mutation operator that mutates a gene in a chromosome by altering its arrival time. The idea behind this operator is to move task executions along $T$, the specified time interval. The aim of this is to find the optimal times at which aperiodic task arrival times will be more likely to cause deadline misses. This is done in such a way that the constraints on the chromosomes are met. Constraints 1 and 2 stipulate that the difference between two consecutive genes’ arrival times is greater than $min_i$ and less than $max_i$. There is one exception: if one - or both - genes have arrival times equal to $-1$. Hence, in mutating a gene, we consider three situations:
1. The gene chosen for mutation is not the first execution, the last execution, or one with an arrival time of –1.
2. The gene chosen for mutation has arrival time equal to –1.
3. The gene chosen for mutation is the first execution of the task. In other words, it is the first gene with arrival time not equal to –1.

In the first situation, when the chosen gene, $j$, lies somewhere in the middle of the first and last task executions, the values its arrival time can take are confined between $a_{j-1} + \min_i$ and $a_{j-1} + \max_i$. However, by altering the arrival time of $j$, the value of $j+1$ might fall outside the permissible range of $[a_{j-1} + \min_i, a_{j-1} + \max_i]$. Hence, it too might have to be altered. Effectively, a gene from a chromosome is randomly selected. A new arrival time is chosen for it from the range $[a_{j-1} + \min_i, a_{j-1} + \max_i]$. New values are also generated for any subsequent executions of the same task that follow if the new value causes other genes to fall outside their permissible minimum and maximum range. For any new values chosen that are greater than $T$, that execution is considered not seeded, its value is set to –1, and the gene is moved to the head of the executions for that task. In this way, constraints 3 and 4 are adhered to. Furthermore, if mutation alters the last execution of the task, a check is performed on the last task execution to ensure that it is a valid ending execution and that no other executions should occur after it. Hence, if the difference between the last execution of the task, $j$, and $T$ is greater than $\max_i$, one of the non-seeded executions are modified with their new values chosen from the range $[a_j + \min_i, a + \max_i]$.

Let us consider an example with three tasks $t_1$ ($\min_{t_1} = 200$), $t_2$ ($\min_{t_2} = 150$, $\max_{t_2} = 200$) and $t_3$ ($\min_{t_3} = 400$) and $T=400$. A sample chromosome composed of six genes (numbered 1 to 6) is:

$$
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
(t1, -1) & (t1, 200) & (t2, 50) & (t2, 200) & (t2, 375) & (t3, 0)
\end{array}
$$

Let us assume that gene 4 is randomly chosen for mutation. Because the gene is not the first in $t_2$’s execution sequence, 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 would now be:
It is important to note that the new value chosen for the gene can also be less than the original value, if the chosen value is within the permissible range. Effectively, this simulates the movement of the task execution upwards, when visualized on a timing diagram. Because gene 4’s value was altered, succeeding genes must be checked. The value for gene 5 should be within the range of \([370, 420]\) \(([220 + 150, 220 + 200])\). The current value of the gene lies within this range. Hence, its value along with any subsequent genes is not altered.

In the second situation, when 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. By eliminating an execution with value equal to –1, the overall effect is the addition of a task execution. When inserting a new task execution, every two consecutive task executions are examined to determine whether an insertion between them will not violate either minimum or maximum interarrival times. If this is the case, the new gene is inserted in that location. Otherwise, the subsequent consecutive task executions are examined. When examining the first non-negative execution of the task sequence, gene \(j\), insertion can only occur before this gene if the value of the arrival time is greater than or equal to the minimum interarrival time of the task. A value greater than or equal to \(\min_i\) indicates that values lying in the range \([0, a_j - \min_i]\) may be inserted before \(j\) while still upholding minimum and maximum interarrival time constraints. Similarly, for consecutive genes \(j\) and \(j+1\), a new gene can be inserted in-between if and only if \(a_{j+1} - a_j \geq 2 * \min_i\). Values lying in the range \([a_j + \min_i, a_{j+1} - \min_i]\) may be inserted between \(j\) and \(j+1\) while still upholding the time constraints. When examining the last gene of the execution sequence, a new gene with values lying in the range \([a_j + \min_i, a_j + \max_i]\) can be inserted after it. Insertions thus occur from left to right along the executions of a task. This means that the new task executions are added whenever they will not violate constraints among already existing executions. If no suitable insertion location is found, this inherently means that no task execution can be added among the already existent task executions. Rather than resorting to backtracking, a different gene altogether is randomly chosen for mutation.
Consider two tasks with \( t_1 \) \((\text{min}_{t_1} = 200)\), \( t_2 \) \((\text{min}_{t_2} = 150, \text{max}_{t_2} = 200)\) and \( t_3 \) \((\text{min}_{t_3} = 400)\) and \( T=400\). A sample chromosome is:

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
(t1, -1) & (t1, 200) & (t2, 50) & (t2, 200) & (t2, 375) & (t3, 0)
\end{array}
\]

Assume gene 1 is the gene that is randomly chosen for mutation. We will 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 \( \text{min}_{t_1} \). Thus, we can insert the new execution before it. The range of choice is \([0, 0]\) \(([0, 200 – 200])\). The mutated chromosome thus becomes:

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
(t1, 0) & (t1, 200) & (t2, 50) & (t2, 200) & (t2, 375) & (t3, 0)
\end{array}
\]

In the third situation, where the gene chosen for mutation is the first execution of the task, a different procedure applies. In this case, the values that gene can take are limited between zero and \( \text{max}_i \). A gene from a chromosome is randomly selected. A new arrival time is chosen for it from the range \([0, \text{max}_i]\). If \( \text{max}_i \) is not specified, the new value is chosen from the range \([0, T]\). Like situation one, new values are also generated for any subsequent executions of the same task that follow if the new value causes other genes to fall outside their permissible minimum and maximum range. Executions with values that are greater than \( T \) are set to –1, and the gene is moved to the head of the executions for that task. If mutation alters the last execution of the task, the check is performed on the last task execution to ensure that it is a valid ending execution, and one of the non-seeded executions are modified with their new values chosen from the appropriate range as stipulated by situation one.

Consider the sample chromosome below based on two tasks with \( t_1(\text{min}_{t_1} = 200)\), \( t_2 \) \((\text{min}_{t_2} = 150, \text{max}_{t_2} = 200)\) and \( t_3 \) \((\text{min}_{t_3} = 400)\) and \( T=400\). Gene 3 is chosen for mutation.

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
(t1, -1) & (t1, 200) & (t2, 50) & (t2, 200) & (t2, 375) & (t3, 0)
\end{array}
\]

A new value is chosen for the chosen gene from the range \([0, 200]\) e.g. 100. The chromosome thus becomes:
We next examine gene 4. Its value now lies outside the permissible range of \([250, 300]\) \(([100+150,100+200]\)). Hence, a new value is chosen for it, e.g. 250. The chromosome now becomes:

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
(t1, -1) & (t1, 200) & (t2, 100) & (t2, 200) & (t2, 375) & (t3, 0)
\end{array}
\]

Gene 5 now lies outside its permissible range of \([400, 450]\) \(([250+150, 250+200]\)). Assume the new value chosen is 420. Because this value is greater than \(T\), it is set to \(-1\) and gene 5 is moved to the head of the t2’s task executions:

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
(t1, -1) & (t1, 200) & (t2, 100) & (t2, 250) & (t2, 375) & (t3, 0)
\end{array}
\]

Gene 5 now becomes the last gene of t2’s executions. We must then check that it is a valid ending execution. \(150 (400-250) < 200\). Hence, this is a valid ending execution. The final mutated chromosome appears as above. It is important to note that because this mutation operator allows task executions to move along the specified time interval, \(T\), in both directions, when visualized on a timing diagram, it effectively explores the solution space. Furthermore, backtracking is eliminated by not committing any changes unless the changes lead to a valid mutation.

Throughout the GA literature, various mutation rates have been used to transform chromosomes. Mutation rates are critical. If the rates are too high, too many good genes of a chromosome are mutated and the GA will stall in converging [18]. Bäck [16] enumerates some of the more common mutation rates used. The author states that De Jong suggests a mutation rate of 0.001, Grefenstette suggests a rate of 0.01, while Schaffer et. al. formulated the expression \(\frac{1.75}{\lambda \sqrt{l}}\) (where \(\lambda\) denotes the population size and \(l\) is the length of the chromosome) for the mutation rate. Mühlenbein suggests a mutation rate defined by \(\frac{1}{\text{length}}\) [17]. Smith and Fogarty in [17] show that, of the common mutation rates, those that take the length of the chromosome and population size into consideration
perform significantly better those that do not. Based on these findings, we apply the mutation rate suggested by Schaffer et. al.:

$$\lambda = \frac{1.75}{\sqrt{l}}.$$

### 5.2.5 Objective Function

Optimization problems aim at searching for a solution within the search space of the problem such that an objective function is minimized or maximized [23]. In other words, the objective function can aim at either minimizing the value of chromosomes or maximizing them. Recall from Section 4.2 that our optimization problem is defined as: what sequence of inputs will cause the greatest delays in the executions of the target task? We identify a number of criteria that define some desirable properties the objective function should possess:

1. **Handles deadline misses:** The objective function must be capable of dealing with deadline misses. The objective function should reward deadline misses. This is, after all, the aim of performance stress testing.

2. **Considers all task executions:** The objective function can adopt one of two approaches: it can either take all task executions into consideration within its calculation, or it can choose one particular execution. The latter approach allows focus to be set on a particular task execution. Hence, a group of bad executions will not be considered, rather only the worst execution. The former approach, on the other hand, considers all task executions. We consider this approach better.

3. **Rewards task executions appropriately:** In real-time systems, one execution that misses its deadline is enough for the whole system to fail. Hence, the objective function should recognize this and ensure that many good executions do not wind up overshadowing one bad execution.

Taking the criteria above into consideration for our optimization problem, we consider the difference between the deadline of an execution and the execution’s actual completion, i.e. $d_{ij} - e_{ij}$. The smaller the difference between deadline and completion, the greater the delay. As seen in Section 4.1, deadlines can be missed - theoretically - yielding negative values for the difference. We are thus interested in rewarding smaller values of the difference and penalizing larger values. Hence, we define an objective
function that assigns scores to each chromosome, \( c \), for a particular target task, \( t \), using equation 8.

\[
f(c) = \sum_{j=1}^{k} 2^{e_{t,j} - d_{t,j}} \quad (8)
\]

It is important to note that the difference between deadline and completion time is reversed in the objective function so that the difference is negative in most cases. The more negative the number, i.e., the greater the difference, the smaller the value of \( f \). In other words, larger values of \( f(c) \) are indicative of fitter individuals. It is also important to note that the objective function is centred on a target task. This allows focus to be centred on the most time critical task to ensure that its deadlines are all met. The objective function is defined for all tasks, not just aperiodic ones. Recall the calculation of \( d_j \) for periodic and aperiodic tasks (Section 5.1):

\[
d_{n,j} = j \tau_i
\]

\[
d_{a,j} = a_{A,j} + d_i
\]

The calculation of \( e_{t,j} \) is scheduler specific. In other words, it is determined by the scheduler. For the scheduler we implement, the calculation of \( e_{t,j} \) uses values from the chromosome as well as values from tasks with higher priorities than \( t \). The completion time of an execution of the target task is a summation of its start time, execution time and time spent in pre-emption by higher priority or dependent tasks. Hence

\[
e_{t,j} = s_{t,j} + C_t + \sum_{y=1}^{p_t-1} h_y \quad (9)
\]

where \( h_y \) is the pre-emption time spent by higher priority task \( y \).

The objective function sums the differences of all the executions of the target task. Summing the differences enhances the survival of fitter chromosomes.

The objective function can alternatively be expressed in minimization format:

\[
f(c) = -\sum_{j=1}^{k} 2^{e_{t,j} - d_{t,j}} \quad (10)
\]

During scheduling, some target task executions’ ending times lie outside the interval defined for testing. Hence, their ending times are not known. As a result, when calculating the difference between task execution end and deadline for the objective score
of a chromosome, these task executions cannot be used. As a result, the objective score sums a set of differences that is smaller than normal. For example, let us assume that a target task has a maximum of four executions and that two chromosomes have the following values for the differences between the target task’s ending times and deadlines \((d_{ij} - e_{ij})\): C1: 3, 15, 16 and C2: 4, 4, 4, 10. Note that C1 has one less difference than C2 because one of the task executions of the target task extends beyond the defined time interval for testing. Using equation 8, the objective score obtained for each chromosome is: C1: \(2^{-3} + 2^{-15} + 2^{-16} \approx 0.1250\) and C2: \(2^{-4} + 2^{-4} + 2^{-4} + 2^{-10} \approx 0.1885\). Although C1 has the smallest difference, three, its objective score is lower than that of C2, indicating that C2 is fitter. C2 is indeed fitter. Because C2 has three task executions that are four time units away from breaking their deadline, and because all execution times are estimates, C2 is more likely than C1 to have one of its task executions not meet its deadline. In other words because all execution times, both those comprising the target task and all other tasks, are estimates, task executions can be longer or shorter than the estimate. This implies that the target task’s executions can be off, or the task executions of higher priority tasks can be inaccurate. Hence, deadlines will more likely be missed at several task executions than in one single task execution. Thus, C2, the chromosome with three equal differences will be three times more likely to miss a deadline than C1 where only one difference may cause a deadline to be missed.

In the example above, we notice that \(2^{-4} + 2^{-4} = 2^{-3}\). In other words, two occurrences of a difference of four are equivalent to one difference of three, as far as the objective function is concerned. Similarly, for other bases, \(3^{-4} + 3^{-4} + 3^{-4} = 3^{-3}\): three occurrences of four are equivalent to one occurrence of three; and \(4^{-4} + 4^{-4} + 4^{-4} + 4^{-4} = 4^{-3}\): four occurrences of four are equivalent to one occurrence of three. Hence, depending on the base to which the differences are raised, the scaling factor, or the number of occurrences of a value equal to one occurrence of the previous value, is determined. With a base of two, two occurrences of a value are equal to one occurrence of the previous value; with a base of 3, three occurrences are equal to one; and with a base of 4, four occurrences are equal to one.

The scaling factor should ideally depend on the average number of task executions. Even then, some means of determining the number of occurrences equal to
one occurrence of the previous value is needed. For our application, we deem a scale of two to be an appropriate value. That is, two occurrences of a value are considered equal to one occurrence of the previous value.

We considered a number of alternative objective functions. These, however, presented various problems. The alternative objective functions along with a discussion of their problems are presented in [32].

**5.3 Prototype Tool**

Following the principles described in the previous sections, we have built Real-Time Test Tool (RTTT), a prototype tool. Real-Time Test Tool is an automated system that identifies potential performance scenarios in any real-time application. The application’s behaviour can be summarized in a sequence of steps: users first input three categories of information: (1) for each task, the task information, comprised of a task number, priority, estimated execution time; dependencies with other tasks (if any) as well as triggers of other tasks (if any); a period for periodic tasks; optional end times for limited periodic tasks as well as minimum and – possibly - maximum inter-arrival times and deadline for aperiodic tasks (2) test environment information comprised of the target task as well as time interval during which testing is to be performed (3) whether the tool should output a timing diagram corresponding to the result. RTTT then generates many sequences of inputs. An objective function determines each sequence’s quality based on a merit value. New sequences are generated and compared. The sequence with the best merit value reached is then reported as well as a measure of safe estimate percentage.

It should be noted that the group of tasks under test, both periodic and aperiodic, may be schedulable under the Generalized Completion Time Theorem and its extension (Section 2.2.2). Recall from Section 4.1 that tasks deemed schedulable by the extension of the GCTT may not be so. If they are schedulable by the GCTT, RTTT will attempt to confirm whether this is really the case and will identify stress test cases. If, on the other hand, they cannot be deemed schedulable by the GCTT and its extension, we do not know whether the tasks are schedulable. We use RTTT to investigate whether we can find a test case where deadline misses occur. If we cannot find such a test case, this does
not guarantee that none exist. However, one can still feel more confident that such a case is unlikely.

The implementation of Real-Time Test Tool can be decomposed into two portions: a scheduler and a genetic algorithm. The scheduler we implement is POSIX compliant as discussed in Section 2.1. As the problem we address is an optimization problem, the underlying engine of our application is a genetic algorithm. The tool we use is GAlib, a C++ library of genetic algorithm objects [14]. GAlib is customized for our application. In depth coverage about the tool is presented in [32].

6 – Case Studies

We report results for a number of case studies, using the principles and prototype tool described in the previous section. The examples selected are representative of typical situations arising from using RTTT. In the first example, we show 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 execution times of the system tasks can then lead to deadline misses. The second example in Section 6.2 is more extreme and less frequent, but nonetheless important, and shows that, even when a set of tasks is deemed schedulable by GCTT, this may not be the case. Assuming execution times are correct, RTTT shows seeding times leading to missed deadlines. In the third example, an avionics application model with hard real-time constraints is run using our tool to demonstrate its applicability to real systems.

In the first two case studies, application of the GCTT and its extension is used as a basis of comparison to assess whether the methodology and RTTT have detected a more stressful test case. In the final case study, the timing diagram depicting the application of the GCTT and its extension is not provided. However, according to the documentation provided by the authors of the system, the tasks used in the case study are schedulable, according to schedulability analysis.

As GAs are a heuristic optimization technique, variances occur in the results produced. To give an idea of the variability of the GA, each case study was run 10 times and the variance in both objective function and difference between execution end and

---

9 Only the major classes of the library are shown. For complete class listing, the reader is referred to [14].
deadline are reported. 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.

6.1 Execution Estimates Case Study

Consider three independent tasks: $t_1$ ($r_{T1} = 255$, $C_{T1} = 200$, $p_{T1} = 32$), $t_2$ ($\min_{T2} = 240$, $C_{T2} = 20$, $p_{T2} = 31$), $t_3$ ($r_{T3} = 250$, $C_{T3} = 20$, $p_{T3} = 30$, $t$). Tasks $t_1$ and $t_3$, the target task, are periodic, $t_2$ is aperiodic, and $t_1$ has a higher priority than $t_2$ and $t_3$. GCCT proves that these three tasks are schedulable (details are provided in [32]). This can be illustrated with a timing diagram as shown in figure Figure 26(a). Note that, in the figure, only relevant parts of the time scale are illustrated.

As specified by the GCCT, the diagram assumes the aperiodic task is transformed into a periodic task and the corresponding triggering event first arrives at time unit 0, along with the two other tasks. The differences between the execution end and the deadline for the target task are 10 and 20 time units respectively (i.e., 250-240 and 500-480, respectively).
Figure 26: Execution estimates example. (a) GCTT execution (b) RTTT execution (c) RTTT execution with estimates

Using RTTT to generate the seeding times, the timing diagram resulting is presented in Figure 26(b). Notice that the second execution of the target task is now 10 time units closer to its deadline than with the GCTT scheduling. Task 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. Its deadline is 500. The difference for the first execution is unchanged.

With a small error increase in execution estimates for each task, namely 4.5%, deadline misses appear in the target task, as illustrated in Figure 26(c). The first executions of the three tasks end at time units 210, 231 and 252 respectively, thus resulting in a missed deadline of 2 time units for t3. This indicates that any inaccuracy greater than 4.5% in execution time estimates will result in missed deadlines.

This case study was run using RTTT a total of 10 times. The results for this case study presented no variance. In all 10 runs, the value of the objective function was
0.00195313 and the largest difference was –10. For the three tasks running on a 500 time unit interval, the average execution time of each run of this case study was one minute.

6.2 Schedulable Tasks Missing Deadlines Case Study

Consider the three tasks: \( t_1 (r_{T1} = 3, C_{T1} = 1, p_{T1} = 32, \text{dependent on } t_3), t_2 (min_{T2} = 8, C_{T2} = 3, p_{T2} = 31), t_3 (r_{T3} = 9, C_{T3} = 2, p_{T3} = 30, \text{dependent on } t_1) \). Tasks \( t_1 \) - the target task - and \( t_3 \) are periodic, \( t_2 \) is aperiodic, and \( t_1 \) has a higher priority than \( t_2 \) and \( t_3 \). Using the GCTT and modeling the aperiodic task as a periodic task with period equal to 8, we can prove that these tasks are schedulable, as [32] illustrates. This can be illustrated with a timing diagram as shown in Figure 27. Figure 27 further assumes that the aperiodic task is ready to execute at time 0, just like the periodic tasks.

In Figure 27(a), \( t_1 \) completes its first execution by time unit 1 (2 time units before its deadline) at which point \( t_2 \) can begin its execution. Task \( t_2 \) is pre-empted by the higher priority task \( t_1 \) at time unit 3, but it is allowed to complete execution after \( t_1 \) has completed. Task \( t_2 \)'s first execution ends at time unit 5, long before its deadline at time unit 8. At this point, \( t_3 \) is now ready to execute and does so. Although the higher priority task \( t_1 \) arrives at time unit 6 while \( t_3 \) is executing, it is dependent on \( t_3 \). Hence, \( t_3 \) is not pre-empted. 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.

![Timing diagram for schedulable tasks. (a) aperiodic tasks modeled as periodic (b) aperiodic tasks not modeled as periodic.](image)

**Figure 27:** Timing diagram for schedulable tasks. (a) aperiodic tasks modeled as periodic (b) aperiodic tasks not modeled as periodic
In [4, 5, 10, 12], 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 constantly arrive at their defined minimum inter-arrival times. Hence, any lower priority tasks will be pre-empted 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, 11 and 19 respectively. These particular arrival times have been produced by our prototype tool. Because of its dependence of 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 Figure 27(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 pre-empts 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.

6.3 Avionics Application Model 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. While representative of such aircraft systems, the performance and complexity requirements did not exactly match those of an actual aircraft. The implementation of the software proceeded from there, with the aid of various software applications [15]. The GAP task set characteristics are presented in Table 4. From the table, only one task is aperiodic, weapon protocol. All other tasks are periodic.
<table>
<thead>
<tr>
<th>Task</th>
<th>Priority</th>
<th>Period (ms)</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer Interrupt</td>
<td>100</td>
<td>1</td>
<td>0.051</td>
</tr>
<tr>
<td>Weapon Release</td>
<td>98</td>
<td>200</td>
<td>3</td>
</tr>
<tr>
<td>Radar Tracking Filter</td>
<td>84</td>
<td>25</td>
<td>2</td>
</tr>
<tr>
<td>RWR Contact Management</td>
<td>72</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>Data Bus Poll Device</td>
<td>68</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>Weapon Aiming</td>
<td>64</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>Radar Target Update</td>
<td>60</td>
<td>50</td>
<td>5</td>
</tr>
<tr>
<td>Navigation Update</td>
<td>56</td>
<td>59</td>
<td>8</td>
</tr>
<tr>
<td>Display Graphic</td>
<td>40</td>
<td>80</td>
<td>9</td>
</tr>
<tr>
<td>Display Hook Update</td>
<td>36</td>
<td>80</td>
<td>2</td>
</tr>
<tr>
<td>Tracking Target Update</td>
<td>32</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>Weapon Protocol</td>
<td>28</td>
<td>Aperiodic</td>
<td>1</td>
</tr>
<tr>
<td>Navigation Steering Commands</td>
<td>24</td>
<td>200</td>
<td>3</td>
</tr>
<tr>
<td>Display Stores Update</td>
<td>20</td>
<td>200</td>
<td>1</td>
</tr>
<tr>
<td>Display Keyset</td>
<td>16</td>
<td>200</td>
<td>1</td>
</tr>
<tr>
<td>Display Status Update</td>
<td>12</td>
<td>200</td>
<td>3</td>
</tr>
<tr>
<td>BET E Status Update</td>
<td>8</td>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>Navigation Status</td>
<td>4</td>
<td>1000</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 4: GAP task set characteristics**

The authors in [15] point out three tasks that make up the weapons system: weapon protocol, weapon aiming and weapon release. The weapon system is activated through an aperiodic event, emulating a pilot button press requesting weapon release. The button press - corresponding to the weapon protocol - triggers the weapon aim task and
waits for another button press to handle a possible abort request. Meanwhile, the weapon aim task periodically computes the release time\(^{10}\) 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 requests are denied and the weapon aim triggers the periodic weapon release task, which must complete its execution within 5 ms of the start of its period. Once the release time is reached, weapon release proceeds to release one weapon every second for a total of five seconds.

The weapons system consists of a number of tasks that are triggered in sequence by one another. In addition, the triggered tasks have limited periods; that is, only execute for a predefined span of time. This task sequencing cannot be managed by considering the events triggering the tasks internal. This is because both triggering and triggered tasks have different priorities. They cannot be combined into one task. The scheduler we had initially implemented did not account for neither this task triggering nor the limited execution time span. Yet, these features were not difficult to incorporate into our application; they merely involved updating our scheduler so that it became capable of handling the aforementioned situations. We stress that no other portion of RTTT was modified with these additions, save the input parser where new tokens were added. Hence, modifications to our implemented scheduling strategy need only create a new scheduler and update the necessary input tokens. With our modification, we added the keywords \texttt{triggers} (to allow tasks to trigger other tasks) and \texttt{end} (to define tasks with limited time spans). For the purpose of this case study, we assume that any input received from the weapon protocol task is the pilot requesting a weapon release action. That is, no requests for aborts or cancels of a previous weapon release are inputted.

Of the set of tasks specified in Table 4, task 1, the timer interrupt task, has non-integer execution time value. We require integer execution time estimates in RTTT. Hence, we are faced with one of two alternatives, rounding the execution estimate value of task 1 up to the nearest integer, or rounding it down to the nearest integer. Rounding up yields an execution estimate of 1 ms. This increase can, however, contribute to target deadline misses when none actually exist. Rounding the timer interrupt task’s execution down to zero would not contribute to the same problem. On the contrary, any deadline

\(^{10}\) Release calculation is not provided by the authors.
misses that appear would reflect the actual situation. The misses would actually be off by a larger value, but they would exist. Thus, we chose to follow the second approach, rounding the value of the task down to zero. We note that, as a result, the differences between target task execution deadline and completion time are actually somewhat larger than those calculated by RTTT.

Furthermore, the weapon release task, as described by the authors in [15] operates with two periods. The task executes at 200 ms intervals from the time it is initiated by the weapon aiming task until the release point. At release time, the task deploys one weapon every second for a total of 5 seconds. To simulate these distinct periods belonging to the same task, we split the weapon release task into two tasks. The first deals with the first period and executes until the release point. It then triggers the second task, weapon release subtask, which executes every second for a total of five seconds at the same priority.

Locke, Vogel and Mesler in [15] do not specify a timing constraint on the overall execution of the weapons system. For the purpose of our case study, we assume that requests for weapon release must be serviced within 1.5 seconds. This is inclusive of the time the request is issued by a pilot and the beginning of the firing of the first of the series of five weapons. Because the notion that the firing of weapons within the specified timing constraints is deemed the most critical task of the tasks described in Table 4, we designate the weapon release task as our target task. The time interval we specify is 15 seconds. This is larger than the 11.8-second time interval specified by the LCM. With 15 seconds, the weapon system can run a maximum of 10 times as opposed to a maximum of 8 in 11.8 seconds. Hence, the time interval used merely allows more requests for the weapon system to be issued.

With the weapon release task as our target, RTTT produces no deadline misses. The closest any of the executions of the target task get to their deadlines is within 197 ms. This is within the time limitation that this task must complete its execution within 5 ms of the start of its period. For this limitation to be met, the least difference between execution end and deadline for the weapon release task is 195. Furthermore, execution estimates will not produce deadline misses if they are accurate within 22.2%. This is a fairly large estimation error, hence the weapon release system will most likely meet its deadline.
Locke, Vogel and Mesler in [15] point out that - based on schedulability analysis - only the first eight tasks described in Table 4 are deemed schedulable. The remaining tasks are not guaranteed to meet their deadlines. Since these are the tasks with the highest priorities, hence probably the most critical, we decided to run RTTTT with each of the remaining seven tasks designated as our target. Despite schedulability analysis claims, we were able to produce sequences that cause some tasks to violate their timing constraints. Our findings are summarized in Table 5 below. The table lists the different target tasks, the seeding times of the aperiodic task generated by our prototype, the execution numbers within the target task that caused deadline misses, the numerical value of the deadline miss in milliseconds, and the percentage under which execution estimates are safe.

From Table 5, 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 airplane. Radar Target Update updates various target positions, while Navigation Update computes aircraft position, altitude and rates. Note that in all these tasks that result in missed deadlines, at least one execution from each task is caused at time units less than 11.8 seconds, the time interval based on LCM. In other words, with $T = 11.8$ seconds, the same tasks in Table 5 would still miss their deadlines. It is important to note that, with the weapon release subtask as target, the completion time limitation is met whereby the closest the executions of this task get to their deadlines is within 997 ms. Hence, each execution of the task does complete within 5 ms of the start of its period.
<table>
<thead>
<tr>
<th>t</th>
<th>a_{weapon protocol,j}</th>
<th>j where e_{t,j}&gt;d_t</th>
<th>Value of Deadline Miss (ms)</th>
<th>Execution Estimation Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weapon Release</td>
<td>4887, 11919, 12122, 14749, 14990</td>
<td>-</td>
<td>-</td>
<td>22.2%</td>
</tr>
<tr>
<td>Weapon Release Subtask</td>
<td>9999, 14066, 14630, 14947</td>
<td>-</td>
<td>-</td>
<td>22.2%</td>
</tr>
<tr>
<td>Radar Tracking Filter</td>
<td>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</td>
<td>-</td>
<td>-</td>
<td>1155.5%</td>
</tr>
<tr>
<td>RWR Contact Management</td>
<td>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</td>
<td>147, 548</td>
<td>3, 9</td>
<td>0%</td>
</tr>
<tr>
<td>Data Bus Poll Device</td>
<td>90, 299, 601, 892, 1093, 1295, 1687, 1896, 2100, 2415, 2802, 3048, 3261</td>
<td>-</td>
<td>-</td>
<td>211.1%</td>
</tr>
<tr>
<td>Poll Device</td>
<td>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</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weapon Aiming</td>
<td>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</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radar Target Update</td>
<td>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</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Navigation Update</td>
<td>157, 368, 579, 781, 998, 1377, 1589, 1816, 2043, 2317, 2593, 2829, 3036, 3376, 3595, 3861, 4108, 4408, 4633, 98, 101,</td>
<td>17, 16, 10, 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>217, 214, 218, 267</td>
<td></td>
<td>22.2%</td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td>RTTT Results</td>
<td>Deadline Misses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------</td>
<td>----------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radar Tracking Filter</td>
<td>118, 132, 135, 190, 245</td>
<td>23, 2, 28, 27, 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Bus Poll Device</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RWR Contact Management</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weapon Aiming</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radar Target Update</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Navigation Update</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: RTTT results of avionics highest priority tasks

The results of multiple runs of this case study produced some variance in the output as shown in Table 6 below. From the table, the first two tasks produce very little variance. Contrary to the results reported in Table 5, Radar Tracking Filter and Data Bus Poll Device result in deadline misses in three and two of the 10 runs respectively. RWR Contact Management and Weapon Aiming, on the other hand, always conform to their deadlines; within a variance of nine and 10 time units respectively. Similarly, Radar Target Update and Navigation Update always produce deadline misses within a variance of four and 12 time units respectively. Because RTTT is built on a GA heuristic engine, this variance is natural. In general, the variance is considered relatively low. However, we would advise that RTTT be run a minimum of 10 times for each target task, to ensure that the underlying GA is not caught in a local minimum, especially when considering that the running time for each task is quite reasonable. For the eight tasks of this case study running on a 15000 time unit interval, the average execution time of each run was 46.5 minutes.
<table>
<thead>
<tr>
<th>Target Task</th>
<th>Run Number</th>
<th>Objective Function Value</th>
<th>Largest Value of e_{T,j} – d_T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weapon Release</td>
<td>1-10</td>
<td>0</td>
<td>-197</td>
</tr>
<tr>
<td>Weapon Release Subtask</td>
<td>1,2,4,6-10</td>
<td>0</td>
<td>-997</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>-994</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
<td>-995</td>
</tr>
<tr>
<td>Radar Tracking Filter</td>
<td>1</td>
<td>0.191298</td>
<td>-3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.116635</td>
<td>-4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>64.002</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.00288</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.0889019</td>
<td>-4</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.0427746</td>
<td>-5</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0.559302</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0685323</td>
<td>-4</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>18.0067</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.130897</td>
<td>-3</td>
</tr>
<tr>
<td>RWR Contact Management</td>
<td>1</td>
<td>520.035</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>112.96</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>32.094</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>552.665</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2049.11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>16664.2</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8640.07</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>136.587</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>403.03</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>162.134</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

**Data Bus Poll Device**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0371494</td>
<td>-5</td>
</tr>
<tr>
<td>2</td>
<td>0.562528</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>0.257822</td>
<td>-2</td>
</tr>
<tr>
<td>4</td>
<td>2.01564</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0.0629904</td>
<td>-4</td>
</tr>
<tr>
<td>6</td>
<td>0.039091</td>
<td>-5</td>
</tr>
<tr>
<td>7</td>
<td>0.00172283</td>
<td>-12</td>
</tr>
<tr>
<td>8</td>
<td>0.000150355</td>
<td>-14</td>
</tr>
<tr>
<td>9</td>
<td>16.0001</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>0.500017</td>
<td>-1</td>
</tr>
</tbody>
</table>

**Weapon Aiming**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0201419</td>
<td>-6</td>
</tr>
<tr>
<td>2</td>
<td>0.00586035</td>
<td>-8</td>
</tr>
<tr>
<td>3</td>
<td>0.00595128</td>
<td>-8</td>
</tr>
<tr>
<td>4</td>
<td>0.126953</td>
<td>-3</td>
</tr>
<tr>
<td>5</td>
<td>0.000214935</td>
<td>-13</td>
</tr>
<tr>
<td>6</td>
<td>0.0195313</td>
<td>-6</td>
</tr>
<tr>
<td>7</td>
<td>0.000489237</td>
<td>-11</td>
</tr>
<tr>
<td>8</td>
<td>0.157288</td>
<td>-3</td>
</tr>
<tr>
<td>9</td>
<td>0.0312508</td>
<td>-5</td>
</tr>
<tr>
<td>10</td>
<td>0.00394637</td>
<td>-8</td>
</tr>
<tr>
<td>Radar Target Update</td>
<td>1</td>
<td>198144</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>105830</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>104858</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>139265</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>140288</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>262146</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>590081</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>163840</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>81921.2</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>24578.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Navigation Update</th>
<th>1</th>
<th>5242880000</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>6442450000</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>18387800000</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>17179900000</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>67148000</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>13851300000</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>72351700</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>14602900000</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>18547300000</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>30924000000</td>
<td>38</td>
</tr>
</tbody>
</table>

**Table 6:** RTTT results of 10 runs
7 – Conclusions

Reactive real-time systems must react to external events within time constraints: Triggered tasks must execute within deadlines. Because real-time systems are often safety critical systems, failures in adhering to time constraints are intolerable. Deadlines that are not met can sometimes lead to life-threatening situations. The risk of this occurring can greatly be reduced through some means of deadline detection misses. This work’s objective is precisely that; to automate, based on the system task architecture, the derivation of test cases that maximize the chances of critical deadline misses. We have shown that deadlines may be missed even though the associated tasks have been identified as schedulable through appropriate schedulability analysis (in our case, the Generalized Completion Time Theorem). Though it is argued that schedulability analysis simulates the worst-case scenario of task executions, this is not always the case as schedulability theory makes a number of assumptions:

1.) The first occurrence of events triggering aperiodic tasks happens at the same time as the first occurrence of events triggering periodic tasks: at the very beginning of the observation period during which task completion times are monitored and analyzed.

2.) The interarrival time between two consecutive arrivals of an event triggering an aperiodic task is always constant.

3.) Schedulability theory assumes that execution times are accurate, when they are mere estimates. Hence, while a group of tasks may be deemed schedulable, deadline misses may actually occur if task execution time estimates turn out to be inaccurate.

These three reasons combined can lead to unrealistic situations since the arrival of events triggering aperiodic tasks is often unpredictable.

Our automation tailors Genetic Algorithms to generate seeding times for aperiodic tasks based on task information, such as estimated execution times and priorities. The generated times aim at maximizing the chances of deadline misses of a specific target task. Users are free to focus on any task they deem critical for the application. Because Genetic Algorithms are a heuristic optimization technique, they do not always necessarily
provide the optimal solution. However, as Section 6 reveals, the solutions provided usually have little variability. In our methodology, we note that variations on the length of the chromosome, the number of tasks, and the testing time interval affect the convergence of the Genetic Algorithm used.

Using Real Time Test Tool (RTTT), we have performed a number of case studies. These case studies cover a wide range of situations by varying the number of tasks and priority patterns. The results of the case studies suggest two important practical points:

1. 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.
2. Even when tasks are schedulable by the GCTT, RTTT can sometimes identify seeding times that will lead to missed deadlines if the execution time estimates are accurate. This may be helpful in identifying performance problems in the early design stages.

We have implemented RTTT in such a way that it is easily adaptable. Changes to the fixed priority, pre-emptive scheduling strategy used need only modify the scheduler in the tool. In one of the case studies performed, we did actually modify the scheduler, allowing for task triggering from other tasks.

This paper provides a practical solution that can help automate the performance stress testing of real-time, reactive systems. Whether during design or testing, this automation is likely to help testers identify response time problems.

We consider the direction of future work in our methodology. While we chose to use steady state GA in our methodology, the different variations on GAs can be used instead and compared. Hence, the methodology and tool can be adapted to run using simple, incremental and deme GAs. Furthermore, the methodology can also be adapted with a different selection strategy.

The scheduler assumed in this paper is fixed priority pre-emptive, with task dependencies in the form of shared resources. This entails that when dependencies occur between tasks, tasks cannot execute until their dependent tasks have fully completed their execution. It is important to note that the scheduler used does not affect the applicability of the methodology we present. The scheduler in our methodology is merely a black box that is used in the calculation of the objective function. Indeed, any commercial scheduler
could have been used, yet none were found to exist. Nevertheless, in the future, other
types of dependencies can be accounted for in the scheduler; ones where tasks await
various pieces of information from dependent tasks.

Similarly, distributed systems can be accounted for in the future within the
scheduler. Distributed systems cannot accurately predict the delays associated with task
triggering. This would affect the calculation of the fitness function. In this case, perhaps
some means of a fitness distribution incorporated into the fitness function would be
needed. The distribution would represent the minimum, maximum and average delay
times for triggering and incorporate them into the task completion calculation.

Several solutions exist for the optimization problem in the field of Artificial
Intelligence. There do not seem to be clear-cut lines as to which technique should be
applied for a given problem. It would be interesting to apply the theory presented in this
paper to another optimization technique such as simulated annealing or tabu search, and
then compare the results that each approach achieves. The various optimization
techniques can be used with the same objective function presented in Section 5.2.5, as it
is independent of Genetic Algorithms.

The limitations on schedulability theory discussed in the paper present a broad
area of future research. Because the extension of the GCTT converts aperiodic tasks to
periodic ones [5, 6, 7, 8], it does not accurately simulate the worst-case scenario of task
arrival times, as demonstrated in Section 6.2. Use of the GCTT with this extension is not
valid. This is because the GCTT assumes that the first aperiodic task arrival is available
for scheduling at the beginning of the observation period, along with the first occurrences
of periodic tasks. Furthermore, the extension of the GCTT maintains constant interarrival
times between the consecutive arrivals of the aperiodic tasks. While this may reflect a
bad scenario of task arrival times, a worse scenario may exist if aperiodic tasks are left as
such – arriving sporadically. The implications of these assumptions are great, since a
group of tasks are sometimes deemed schedulable when at least one scenario exists when
they are not. This in turn stimulates an area of research that tries to find better modelling
techniques for aperiodic tasks in schedulability analysis. This is because treating
aperiodic tasks as periodic ones does not always represent the worst-case scenario. At
best it is a good heuristic, but it is not a guarantee that the associated tasks are schedulable.
References


