Performance Stress Testing of Real-Time Systems Using Genetic Algorithms

By

Marwa Shousha

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfilment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

December 2003

Copyright © 2003 by Marwa Shousha

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-93946-4

Our file Notre référence

ISBN: 0-612-93946-4

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

PAGINATION ERROR.

ERREUR DE PAGINATION.

TEXT COMPLETE.

LE TEXTE EST COMPLET.

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.

Acknowledgements

All praise be to Allah, the Benevolent, the Merciful, without whose grace and blessing this work would not have been possible.

I would like to thank my supervisors, Dr. Briand and Dr. Labiche, for their guidance, direction and support. Their insight and ideas were very constructive, but I owe them much more for showing me the proper way to conduct research. The long meetings were not in vain, after all! To them I dedicate a special “thank you”.

I would also like to thank my defence committee members, Dr. Williams, Dr. Woodside and Dr. Majumdar. Their comments and suggestions were very helpful, and made the discussion very enjoyable. I would also like to thank them for nominating me for the Senate Medal.

To my parents, none of this would have been possible were it not for your prayers. This work is dedicated to you.

Finally, I would like to thank my husband, Abdelrahman, and daughter, Aya, for brightening up my life and being there for me. To my husband, I only hope I can be as supportive in your thesis.

Table of Contents

Chapter 1 - Introduction	1
1.1 Problem Description.....	3
1.2 Thesis Contribution.....	6
1.3 Thesis Organization.....	6
Chapter 2 – Background	8
2.1 Processor Execution	8
2.2 Task Scheduling Strategies	9
2.3 Schedulability Theory	11
2.3.1 Rate Monotonic Algorithm	11
2.3.2 Generalized Completion Time Theorem.....	12
Chapter 3 – Related Work.....	16
Chapter 4 – Real-Time Test Theory.....	20
4.1 Limitations of Schedulability Theory	20
4.2 No Simple Strategy	27
4.2.1 Example 1.....	27
4.2.2 Example 2.....	31
4.2.3 Summary	35
4.3 Genetic Algorithms	36
4.3.1 Moving to Genetic Algorithms	36
4.3.2 Genetic Algorithm Overview	38
Chapter 5 -Tailored GA.....	43
5.1 Notation.....	43
5.2 GA Components.....	46
5.2.1 Chromosome	48
5.2.2 Chromosome Length.....	51
5.2.3 Chromosome Initialization.....	54
5.2.4 Operators	55
5.2.4.1 Crossover Operator	56
5.2.4.2 Mutation Operator	59
5.2.5 Objective Function	65
Chapter 6 – Real-Time Test Tool.....	70
6.1 Scheduling Strategy.....	71
6.2 GA Tool - GALib	73
6.3 GALib Customization.....	76
6.3.1 Chromosome	77
6.3.2 Chromosome Initialization.....	78

6.3.3 Operators	78
6.3.3.1 Crossover Operator	78
6.3.3.2 Mutation Operator	79
6.4 User Inputs	79
6.4.1 Input File Format.....	81
6.4.2 Command-line Options	82
6.5 Application Outputs	82
Chapter 7 – Case Studies	85
7.1 Execution Estimates Case Study	86
7.2 Schedulable Tasks Missing Deadlines Case Study	88
7.3 Avionics Application Model Case Study.....	90
Chapter 8 – Conclusions	102
References	107
Appendix A – Sample Input File.....	110
Appendix B – Sample Output File.....	111
Appendix C – Proof of Schedulability of Execution Estimates Case Study.....	114
Appendix D – Enhanced Metamodel.....	116
Appendix E - Pseudocode	118
Appendix F – GALib Patch.....	136
Appendix G – Proof of Schedulable Tasks Missing Deadlines Example	137
Appendix H – Alternative Representations	140

List of Figures

Figure 1: <i>Timing Diagram Example</i>	15
Figure 2: <i>Multimedia Petri net example</i>	17
Figure 3: <i>Reachability graph of multimedia object</i>	18
Figure 4: <i>Timing diagram with aperiodic task modelled as periodic</i>	22
Figure 5: <i>Timing diagram with aperiodic task not modelled as periodic</i>	24
Figure 6: <i>Timing diagram illustrating estimates</i>	25
Figure 7: <i>Timing diagram with 2% estimates</i>	26
Figure 8: <i>Benchmark for Example 1</i>	28
Figure 9: <i>Application of the GCTT</i>	29
Figure 10: <i>Triggering with target task</i>	30
Figure 11: <i>Triggering with highest priority task</i>	31
Figure 12: <i>Benchmark for Example 2</i>	32
Figure 13: <i>Application of the GCTT</i>	32
Figure 14: <i>Triggering with next highest priority</i>	33
Figure 15: <i>Triggering before end of target task</i>	34
Figure 16: <i>GA chromosome terminology</i>	39
Figure 17: <i>Illustration of crossover concept</i>	40
Figure 18: <i>Illustration of mutation concept</i>	40
Figure 19: <i>Flow chart of genetic algorithm (Adapted from [18])</i>	41
Figure 20: <i>Illustration of task execution definition</i>	44
Figure 21: <i>Metamodel of relationships</i>	50
Figure 22: <i>Total number of executions example</i>	52
Figure 23: <i>LCM example</i>	53
Figure 24: <i>Illustration of crossover operators</i>	58
Figure 25: <i>Basic GALib hierarchy</i>	75
Figure 26: <i>Execution estimates example</i>	87
Figure 27: <i>Timing diagram for schedulable tasks</i>	89
Figure 28: <i>Enhanced Metamodel</i>	116

List of Tables

Table 1: <i>Summary of terms</i>	46
Table 2: <i>Application inputs</i>	80
Table 3: <i>Input keywords</i>	82
Table 4: <i>GAP task set characteristics</i>	91
Table 5: <i>RTTT results of avionics highest priority tasks</i>	97
Table 6: <i>RTTT results of 10 runs</i>	101

Chapter 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 2.3.2), 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.

1.2 Thesis Contribution

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.

1.3 Thesis Organization

Chapter 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 Chapter 3, which presents a survey of related work. Chapter 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. Chapter 5 describes how the Genetic Algorithm was tailored. The tool developed is presented in detail in Chapter 6, along with the scheduling strategy implemented. Case studies are then reported in Chapter 7. We draw conclusions and present ideas for future work in Chapter 8. Appendix A presents a sample Real Time Test input file, Appendix B illustrates a sample

output of the tool, and Appendix C provides the schedulability proof of the execution estimates case study. Appendix D presents the pseudocode of the tailored Genetic Algorithm, Appendix F presents a patch for the Genetic Algorithm library used, and Appendix G provides the proof for the schedulable tasks missing their deadlines example. Finally, Appendix H presents alternatives to the various representations used in the Genetic Algorithm.

Chapter 2 – Background

This chapter familiarizes the reader with concepts that are used throughout this thesis. We introduce different processor execution concepts (Section 2.1), task scheduling strategies (Section 2.2) and schedulability theory (Section 2.3). Readers familiar with these concepts may wish to move on to Chapter 3.

2.1 Processor Execution

Applications can run on one of two types of systems: simple or distributed. In a simple system, the application runs in an environment consisting of a single node, usually a single processor. A distributed system, on the other hand, executes in an environment consisting of multiple nodes. The nodes may be in geographically different locations [5], multiple processors on the same machine, or multiple threads on the same processor. The difference between the two system types, simple and distributed, lies mainly in the implementation of the scheduler. In a simple system, the scheduler would need to identify the different tasks that are ready to execute and then proceed to execute them. Similarly, in a distributed system, the scheduler would determine the tasks ready to execute. Additionally, the scheduler would be responsible for task assignment across nodes. In other words, tasks would first be assigned to nodes for scheduling, before determining which tasks are ready to run in each node's environment. Task assignment would need to account for various processor configurations; for example, identical versus heterogeneous processors, which have different task loads [4].

2.2 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 2.3.1) 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].

2.3 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.

2.3.1 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)

2.3.2 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, τ_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 Theroem is:

A set of n periodic tasks can be scheduled by the Rate Monotonic Algorithm¹ for all task phasing if the following condition is satisfied:

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

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

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 ($1 \leq k \leq i$),

$l = 1, \dots, \left\lfloor \frac{T_i}{T_k} \right\rfloor$ 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

¹ The Rate Monotonic Algorithm is extended to deal with priority inversion through the Generalized Completion Time Theorem.

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 $l=1, \dots, \left\lfloor \frac{10}{10} \right\rfloor = 1$. 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 $k=2$. $l_{k=1} = 1, \dots, \left\lfloor \frac{20}{10} \right\rfloor = 1, 2$. We first see whether using $l=1$

will yield a true inequality. Substituting we obtain $1\left(\frac{1}{10}\right)\left\lfloor \frac{10}{10} \right\rfloor + \frac{2}{10} + 0 \leq 1$, i.e. $1 + 2 \leq 10$.

Again, this is a true statement, meaning that t_2 will be able to meet its deadline before the 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].

Figure 1: Timing Diagram Example

Chapter 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: $\alpha\text{VideoB} = \beta\text{VideoA} + 4$ (where αVideoB and βVideoA 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.

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.

Figure 3: Reachability graph of multimedia object

From Figure 3, two possible firing sequences are: $\{ \langle (\text{selDemo}, t_1), (\text{cxlDemo}, t_2), (\text{selFull}, t_3), (\text{cxlFull}, t_4), (\text{exit}, t_5) \rangle \}$, where $0 < t_1 < t_2 < t_3 < t_4 < t_5$ and $\{ \langle (\text{selDemo}, t_1), (\text{cxlDemo}, t_2), (\text{exit}, t_3) \rangle, \langle (\text{selFull}, t_4), (\text{cxlFull}, t_5), (\text{exit}, t_6) \rangle \}$, 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.

Chapter 4 – Real-Time Test Theory

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 4.3 concludes the chapter 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 2.3.2) 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 Appendix G.

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.

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 slightly², setting its arrival times at time units 2, 11 and 19, t_1 – because of its dependence on t_3 – 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 t_2 . They present a scenario that is worse than treating the aperiodic task as periodic. By triggering t_2 at time unit 2, t_3 is allowed to begin its execution for one time unit. When t_2 pre-empts t_3 and t_2 begins its execution, t_1 cannot pre-empt it at time unit 3 because of its dependency on t_3 . Hence, t_1 can only begin its execution once t_3 has completed its execution. However, t_3 completes its execution at the deadline of t_1 , hence causing t_1 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 6.4). The input file is included in appendix A.

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 2.3). If each execution time estimate is off by just 2%, deadlines are missed, as shown in Figure 7.

Figure 7: *Timing diagram with 2% estimates*

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 are³:

* t1: C=1 T=3 P=32
t2: C=1 T=4 P=31
(T) t3: C=1 T=5 P=30

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*

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

³ Based on an example that appears in [10]

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*

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 3. 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*

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*

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 are⁴:

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.

⁴ Based on an example that appears in [5]

Figure 12: *Benchmark for Example 2*

B: GCTT

Applying the worst-case scenario of the Generalized Completion Time Theorem and its extension yields the timing diagram of Figure 13.

Figure 13: *Application of the GCTT*

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.

Figure 14: *Triggering with next highest priority*

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.

Figure 15: *Triggering before end of target task*

In Figure 15, t_3 's execution would have completed at time unit 130. Hence, following the approach, t_2 is triggered at time unit 129 as depicted. Task t_2 completes its execution without pre-emption, allowing t_3 to complete its execution at time unit 160. Once again, as with the Generalized Completion Time Theorem approach, t_3 '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.

4.3 Genetic Algorithms

A number of techniques exist for solving optimization problems. In Section 4.3.1, we present an overview of these methods, along with the reasons behind our choice: Genetic Algorithms. We present an overview of Genetic Algorithms in Section 4.3.2.

4.3.1 Moving to Genetic Algorithms

A variety of methods exist for solving optimization problems. Perhaps the most common techniques are linear and global optimization solution techniques. In linear optimization, or linear programming as it is more commonly known, the objective function as well as all constraints are linear functions of the decision variables to be solved. Linear programming solutions are optimal as they lie on a flat region, namely at the corner of the intersection of the constraints. Global optimization solutions, also known as metaheuristic solutions, continually search for better solutions by altering a set of current solutions [23]. The solutions lie on an uneven solutions space, characterized by multiple peaks and valleys. These peaks and valleys can result in locally optimal solutions; ones where no other solutions in the vicinity have better solutions. Global optimization solutions aim at avoiding local optimal solutions, reaching global ones instead. Simulated annealing, tabu search and genetic algorithms are all global optimization solutions [25].

For the scheduling problem at hand, the number of tasks is not set. Rather, it differs from application to application. As the number of tasks increases, the different combinations representing solutions grows exponentially. As a result, linear programming cannot be used, as combinational explosion would result [24]. Furthermore, for the scheduling problem at hand, any changes in task numbers or execution times may cause great changes in the solution. The solution space of the problem is thus uneven, characterized by multiple peaks and valleys. A global optimization solution is thus needed.

Simulated Annealing (SA) is a technique modeled after the metal annealing process in which metals are melted at high temperatures and allowed to cool slowly. This deliberate cooling process allows the particles composing the solid to align in a regular lattice structure, hence rendering the metal stronger than it was. If the cooling process occurs too quickly, the particles form irregular alignments, resulting in a highly unstable structure [23]. Similarly, Genetic Algorithms (GA) are based on concepts adopted from evolutionary theory. GAs involve a search from a population of solutions rather than a single solution like SAs. With each iteration of the GA, solutions with the highest fitness are recombined and mutated, and solutions with the lowest scores are eliminated. On the other hand, tabu search avoids cycles by penalizing moves that take the solution to points previously visited in the solution space [26].

According to global optimization literature, GAs and SAs are very similar. Some studies, such as [27] indicate that SAs outperform GAs, while others, such as Mann and Smith [28] claim that GAs produce solutions equivalent to or superior to SAs. Most researchers, however, seem to agree that because genetic algorithms maintain a

population of possible solutions, they have a better chance of locating the global optimum compared to simulated annealing and tabu search which proceed one solution at a time [29, 30]. Furthermore, because SAs maintain only one solution at a time, good solutions may be discarded and never regained if cooling occurs too quickly [30]. Similarly, tabu search may miss the optimum solution. Alternatively, steady state GAs, one of the variations of GAs, accept newly generated solutions if they are fitter than previous solutions. Furthermore, genetic algorithms lend themselves to parallelism. Because they manipulate whole populations with both mutation and crossover operators, they can readily be implemented on multiple processors. Simulated Annealing, on the other hand, cannot easily run on multiple processors because only one solution is constantly manipulated [30]. Hence, we adopt genetic algorithms as our optimization solution methodology. With GAs, we gain the flexibility of implementing our solution across multiple processors.

4.3.2 Genetic Algorithm Overview

In [12, 13, 14, 19], the authors describe Genetic Algorithms (GA) as a means of solving complex optimization problems that are often NP hard [13] in limited amounts of time. Optimization problems are those that try to reach the best solution given measurements of the goodness of solutions [19]. GAs are based on concepts adopted from genetic and evolutionary theories. They are comprised of several components: a representation of the solution, referred to as the *chromosome*, fitness of each chromosome, referred to as the *objective function*, the genetic operations of *crossover* and *mutation* which generate new offspring, and selection operations which choose offspring fit for survival.

The chromosome models the problem solution. Each element within the chromosome is known as a *gene*. The collection of chromosomes used by the GA is dubbed *population*. Figure 16 illustrates these concepts in terms of representation of the RGB makeup of a population of three pixels on a screen. The chromosome in the figure is composed of three genes. Each gene represents the red, green, or blue component of a pixel on a screen. Hence, the chromosome depicts one pixel's RGB makeup. The population portrays the makeup of three pixels on the screen.

Figure 16: *GA chromosome terminology*

The quality of a chromosome is its fitness. Fitness defines which chromosomes are closer to the optimal solution. If the optimal solution for the population of Figure 16 is a pixel with only a red component (i.e. a chromosome with RGB values 1, 0, 0), the first two chromosomes of the population would be deemed fitter than the last one. The initial two chromosomes each have one more color component in addition to the red component, while the last chromosome has no red component.

Both crossover and mutation operators are needed to explore the problem search space. Crossover operators generate offspring from two parents based on the merits of each parent, as demonstrated in Figure 17 through single point crossover. Taking the G component as a division point common to both parents, the parents alternate genes with

respect to the division point in creating the children. Parent 1 contributes the RB components of Child 1, allowing Parent 2 to contribute the G component. Similarly, Parent 2 contributes the RB components of Child 2, while Parent 1 contributes its G component. Hence, GAs use the notion of survival of the fittest by passing superior traits from one generation to the next.

Figure 17: *Illustration of crossover concept*

Mutation operators *mutate*, or alter a single chromosome as Figure 18 shows. Mutation aids the GA in avoiding local minima. In the figure, the Red gene is mutated, resulting in a chromosome with RGB values 0, 1, 0 respectively.

Figure 18: *Illustration of mutation concept*

The process of selection determines which individuals among the original population, mutated and child chromosomes will survive, hence retaining a constant population size.

The Genetic Algorithm first randomly creates an initial population of individuals. Working with the population, the genetic algorithm then selects and performs various

crossover and mutation operations, creating new chromosomes. The fitness of the new chromosomes is compared to others in the population. Fitter individuals are retained while less fit ones are removed. The process of crossover, mutation, fitness comparison and replacement continues until the termination criterion is reached. In most cases, the termination criterion is a particular number of runs or generations of the algorithm [14].

Figure 19: *Flow chart of genetic algorithm (Adapted from [18])*

A variety of replacement methodologies are defined for GAs, such as simple, steady state and incremental. Each specifies how much of the population should be replaced with each run or generation of the algorithm. The simple genetic algorithm creates an entirely new population of chromosomes with each generation of the

algorithm. The steady state algorithm, on the other hand, uses overlapping populations, leaving it up to the user to determine the number of chromosomes to replace in each generation. Each generation, the steady state GA produces offspring, storing them in a temporary location. These are then added to the population and the worst individuals are removed such that the population size remains constant. In incremental genetic algorithms, only one or two offspring chromosomes are generated. These are integrated into the population in one of the following ways: replacing the parent, replacing a random individual in the population, or replacing an individual that is similar to the offspring.

Chapter 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 chapter 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 thesis, 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 t1 (period = 50) and t2 (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 t1 has three executions: $a_{1,1} = 0$, $a_{1,2} = 50$ and $a_{1,3} = 100$. For t2, $a_{2,1} = 5$, $a_{2,2} = 40$ and $a_{2,3} = 80$. It is important to note that in $P_{2,2}$, t2 is pre-empted by t1 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

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 t1 with $a_{1,1} = 0$ has its deadline at time unit 50, which marks the period of t1,

$a_{1,2} = 50$ 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.

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

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.

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

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

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 2.3.2). 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 $t1$ ($min_{t1} = 10, C_{t1} = 3, p_{t1} = 32$) and $t2$ ($min_{t2} = 11, C_{t2} = 5, p_{t2} = 31$). In a time interval of 30, the following is a valid chromosome: $(t1, -1) (t1, 19) (t1, 29) (t2, -1) (t2, -1) (t2, 10)$. The chromosome meets all constraints.

The metamodel describing the relationships between the various concepts is presented in Figure 21.

Constraints:
context Chromosome:
self.gene.taskNumber->asSet->forAll(i:int| not(self.PeriodicTask.taskNumber->asSet->includes(i)))

Figure 21: Metamodel of relationships

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 Appendix H.

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

$$k = \left\lceil \frac{T}{\min_i} \right\rceil \quad (6)$$

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. $a_{i,j} = (j-1) \min_i$). For example, consider the aperiodic task set⁶ 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.

⁶ Adapted from an example in [5]

Figure 22: *Total number of executions example*

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

$$l = \sum_{i=1}^n \left\lceil \frac{T}{\min \alpha_i} \right\rceil \quad (7)$$

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: $\lceil 200/100 \rceil + \lceil 200/150 \rceil + \lceil 200/200 \rceil = 2+2+1= 5$. 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 Appendix H.

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; t1, with the higher priority, has a period of four time units and t2 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 t1 and t2 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 Multiple⁷ (LCM) of the periods of all tasks.

Figure 23: *LCM example*

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

⁷ The Least Common Multiple (LCM) of two numbers a and b is the smallest number m for which there exists positive integers n_a and n_b such that $n_a a = n_b b = m$.

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 5.1 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: $[a_{i,j-1} + \min\alpha_i, a_{i,j-1} + \max\alpha_i]$. If there is no previous gene, or the previous gene is not triggered (i.e. $a_{i,j-1} = -1$), the range is $[0, \max\alpha_i]$. 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 t_1 ($\min_{t_1} = 10$, $C_{t_1} = 2$, $\rho_{t_1} = 32$) with $T = 30$. The value of k_1 (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 $[0, \max_1]$ (where because $\max_1$ 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 $[25, 45]$ ($[a_{1,1} + \min\alpha_1, a_{1,1} + \max\alpha_1]$). 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 $k_{i1} + k_{i2} + \dots + k_{i{n-1}}$ 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 *nPointProbCrossover*, the same alternation pattern occurs as *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 tasks⁸ t_1 ($min_{t_1} = 100$, $C_{t_1} = 20$, $p_{t_1} = 32$), t_2 ($min_{t_2} = 150$, $C_{t_2} = 30$, $p_{t_2} = 31$) and t_3 ($min_{t_3} = 200$, $C_{t_3} = 90$, $p_{t_3} = 30$). Because the number of aperiodic tasks is three, two-point crossover is applied dividing the parents as shown in Figure 24.

⁸ Based on an example in [5]

Figure 24: Illustration of crossover operators

In *nPointCrossover*, the fragments of Parent 1 and Parent 2 are alternately interchanged. Using the same example for *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 $[a_j + \min_i, a_j + \max_i]$.

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:

1	2	3	4	5	6
(t1, -1)	(t1, 200)	(t2, 50)	(t2, 200)	(t2, 375)	(t3, 0)

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:

1	2	3	4	5	6
(t1, -1)	(t1, 200)	(t2, 50)	(t2, 220)	(t2, 375)	(t3, 0)

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 ($min_{t1} = 200$), t_2 ($min_{t2} = 150$, $max_{t2} = 200$) and t_3 ($min_{t3} = 400$) and $T=400$. A sample chromosome is:

1	2	3	4	5	6
(t1, -1) (t1, 200)		(t2, 50) (t2, 200) (t2, 375)		(t3, 0)	

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 min_{t1} . Thus, we can insert the new execution before it. The range of choice is $[0, 0]$ ($[0, 200 - 200]$). The mutated chromosome thus becomes

1 2 3 4 5 6

(t1, 0) (t1, 200)	(t2, 50) (t2, 200) (t2, 375)	(t3, 0)
-------------------	------------------------------	---------

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 max_i . A gene from a chromosome is randomly selected. A new arrival time is chosen for it from the range $[0, max_i]$. If 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 $t1(min_{t1} = 200)$, $t2 (min_{t2} = 150, max_{t2} = 200)$ and $t3 (min_{t3} = 400)$ and $T=400$. Gene 3 is chosen for mutation.

1 2 3 4 5 6

(t1, -1) (t1, 200)	(t2, 50) (t2, 200) (t2, 375)	(t3, 0)
--------------------	------------------------------	---------

A new value is chosen for the chosen gene from the range $[0, 200]$ e.g. 100. The chromosome thus becomes:

1 2 3 4 5 6

(t1, -1) (t1, 200)	(t2, 100) (t2, 200) (t2, 375)	(t3, 0)
--------------------	-------------------------------	---------

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:

1	2	3	4	5	6
(t1, -1) (t1, 200)		(t2, 100) (t2, 250) (t2, 375)			(t3, 0)

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 t_2 's task executions:

1	2	3	4	5	6
(t1, -1) (t1, 200)		(t2, -1) (t2, 100) (t2, 250)			(t3, 0)

Gene 5 now becomes the last gene of t_2 '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, Grefensette suggests a rate of 0.01, while Schaffer et. al. formulated the expression $\frac{1.75}{\lambda\sqrt{l}}$ (where λ denotes the population size and l is the

length of the chromosome) for the mutation rate. Mühlenbein suggests a mutation rate defined by $\frac{1}{length}$ [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.: $\frac{1.75}{\lambda\sqrt{l}}$.

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. $d_{i,j} - e_{i,j}$. 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.

$$f(c) = \sum_{j=1}^{k_t} 2^{e_{i,j} - d_{i,j}} \quad (8)$$

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):

$$d_{p_i,j} = jr_i$$

$$d_{A_i,j} = a_{A_i,j} + d_i$$

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

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

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:

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

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 ($d_{t,j} - e_{t,j}$): 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 Appendix H.

Chapter 6 – Real-Time Test Tool

Following the principles described in the previous chapter, 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 2.3.2). 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 6.1. 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 manages the initialization, crossover, mutation, fitness comparison and replacement operations described in Section 4.3.2. GALib is customized for our application. Consequently, we use a steady state genetic algorithm and develop the initialization procedure, mutation operators, crossover operators and fitness function. We also define a chromosome suitable to our application.

6.1 Scheduling Strategy

Real-time systems run on a variety of different platforms (Section 2.2), with a number of different scheduling strategies. We assume one of the more common scheduling strategies based on the POSIX standard: a fixed pre-emptive priority scheduler, where priorities are defined by the designer or developer. The highest priority task executes first, pre-empting lower priority tasks that are executing. It is also worth noting that the underlying schedulability theory used in our approach, the Generalized Completion Time Theorem (Section 2.3.2), also assumes a priority pre-emptive scheduling algorithm [5]. Task priorities range from one, the lowest priority, to 32, the highest priority. Priorities remain static throughout task execution. In other words, during

no point in a task's execution will its priority change. This can, however, be adapted by changing the scheduler. Task dependencies in our application are in the form of shared resource dependencies [5]. If a dependency occurs between tasks, the first ready task of the dependency must fully complete its execution before the dependent task can run, regardless of its priority. Equal priority tasks are executed in a first-come-first-served fashion. Task execution time estimates are integers.

Tasks may trigger the execution of other tasks. Hence, at the end of execution of a task, it can trigger another task with a different priority. Furthermore, periodic tasks may run for predefined intervals of time. That is, they may execute for a limited time span then cease.

The scheduler we implement emulates single processor execution. Assuming distribution would entail the implementation of new scheduler, capable of task assignment across processors.

It is important to note that the assumptions presented are not restrictive. In other words, these assumptions can seamlessly be adapted in our application to meet a wider set of requirements. Modifications to the scheduling strategy would only require the modification of the scheduler. Any new tokens added to the input would need to be updated in the input parser. This is because 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. Hence, various extensions to dependencies can be accounted for in the scheduler; ones where tasks await various pieces of information from dependent tasks.

6.2 GA Tool - GAlib

The tool used to manage the genetic algorithm is GAlib. GAlib was developed by Matthew Wall at the Massachusetts Institute of Technology. GAlib is a library of C++ objects. The library includes tools for using genetic algorithms to do optimization in any C++ program using any representation and any genetic operators [14]. The library has been tested on multiple platforms, specifically DOS/Windows, Windows NT/95, MacOS and UNIX. It can also be used with Parallel Virtual Machines (PVM) to evolve populations in parallel on multiple CPUs [14].

GAlib defines many options. It supports four types of genetic algorithms: simple, steady state, incremental and deme. The former three types are described in Section 4.3.2. The deme genetic algorithm evolves multiple populations in parallel using a steady state algorithm. During each generation, some individuals are migrated between the populations [14]. GAlib also supports various selection methods for choosing an individual for mutation and crossover. These include rank selection, roulette wheel, tournament, stochastic remainder sampling, stochastic uniform sampling and deterministic sampling. In rank selection, the best individual in the population is always chosen [14]. Roulette wheel selection can be conceptualized as a wheel with various slots. Each chromosome occupies a slot whose size is proportional to its fitness. The greater the fitness, the larger the slice. The wheel is then spun and the chromosome corresponding to the chosen slot is selected [19]. Tournament selection uses the roulette wheel to select two chromosomes, and then picks the fitter of the two. In stochastic remainder sampling, the fitness of each chromosome's expected offspring is first evaluated. Then, a temporary population is created with the fitter individuals. Any

fractions resulting from the fitness of the expected representations are used to give that chromosome more likelihood of filling a space. Similarly, in deterministic sampling, a temporary population is created with the individuals whose expected representations are fitter. Any remaining positions are filled by the fittest individuals according to the decimal values of their expected representations. Stochastic uniform sampling randomly picks an individual from the population, giving each individual a probability equivalent to 1 divided by the population size [14]. Replacement strategies include replace parent, replace random and replace worst, for replacing an individual in the population [14]. Built in termination criteria include convergence to a particular fitness value (i.e. users must have an idea of the optimum solution) and number of generations. GALib defines chromosomes that are based on four data types: strings, arrays, lists and trees. Users can additionally define their own chromosome data type. The library also defines default initialization, mutation and crossover operators [14]. Figure 25 illustrates the basic GALib hierarchy⁹.

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

Figure 25: Basic GALib hierarchy

From the figure, `GAScalingScheme` implements the various scaling schemes. The scaling scheme transforms the objective score for each chromosome to a fitness score that the genetic algorithm uses for selection. `GASElectionScheme` implements the selection schemes discussed above. Similarly, `GAGeneticAlgorithm` implements the various flavours of genetic algorithms. It uses each of `GAScalingScheme` and

`GASelectionScheme` in its implementation. It additionally uses `GAGenome`, which represents a chromosome in the library.

6.3 GALib Customization

We tailored the different GALib options to meet our application's needs, as described in Chapter 5. GALib is flexible enough to incorporate all the previously made choices. Of the four supported types of GAs, simple, steady state, incremental and deme, we use steady state GAs. Recall that the steady state algorithm uses overlapping populations, leaving it up to the user to determine the number of chromosomes to replace in each generation (Section 4.3.2). Because a steady state GA replaces a percentage of the population, it ensures that only fitter solutions will replace existing solutions. With simple GAs, the whole population is replaced, hence increasing the likelihood of the replacement of a fit individual. Incremental GAs are relatively slow in converging as they allow the replacement of one or two individuals in the population. Alternatively, we could have chosen deme GAs. We chose steady state instead as it is more commonly known. In a steady state GA, if too few chromosomes survive from generation to generation, the genes available for offspring are limited. On the other hand, if too many chromosomes survive, undesirable genes are increasingly passed on to new generations, as chromosomes with poor traits are not eliminated. The replacement percentage we apply is 50%. This complies with the findings of Haupt and Haupt [18] who present data that shows that GAs perform best when half or less of the chromosomes are kept from generation to generation.

GAlib also supports various selection methods for choosing an individual for mutation and crossover. Of the six supported selection methodologies, we chose the more common roulette wheel selection method.

The actual constituents of the GA from chromosome, operators and objective function were also adapted to fulfill the aims of our application. We created new classes and defined our own hierarchy that integrates with GAlib. Our complete metamodel is presented in Appendix D. This is an enhancement of the metamodel presented earlier in Figure 21. Due to compilation problems, slight modifications to the code constituting GAlib were made. The patches needed to fix these problems are included in Appendix F.

It is important to note that all the values chosen for the mutation rate, crossover rate, population size, number of generations and replacement percentage were tested to ensure that they indeed enhance the performance of the GA. For the stopping criterion, we ran a number of tests on the application with various values. We found the best value to be 500.

6.3.1 Chromosome

GAlib defines a set of chromosomes whose genes are based on four data types: strings, arrays, lists and trees [14]. However, as we need to encode both task number and arrival time, none of GAlib's data types for chromosomes can be utilized. Instead, we derive our own representation (as described in Section 0) based on the library's array chromosome. In pseudocode, this is represented as:

```
Chromosome: array(Gene)
```

where Gene is a structure with two members:

```
struct Gene {  
    taskNumber : int  
    arrivalTime : int
```

}

6.3.2 Chromosome Initialization

GAlib defines default initialization for each of its defined chromosomes [14]. However, as the chromosome in our application is itself derived, we define our own initialization methodology as described in Section 5.2.3. The pseudocode of the initialization process is presented in Appendix D.

6.3.3 Operators

GAlib does not implement backtracking as such. Hence, we make use of our internal backtracking mechanism described in Section 5.2.4.

6.3.3.1 Crossover Operator

GAlib defines default crossover operators for each of the types it defines. However, because we define our own data type, we create our own crossover operator. Our choice of crossover type is somewhat limited by GAlib which only supports sexual and asexual reproduction. As asexual reproduction does not tend to alter genes from generation to generation, we opt to use sexual crossover.

Recall from Section 5.2.4.1 that we define two crossover operators. These are implemented as `nPointCrossover` (crossover without the coin flip) and `nPointProbCrossover` (crossover with the coin flip). We performed 13 trials alternating between the two crossover operators. In each trial, the same test case was run once with `nPointProbCrossover` and once with `nPointCrossover`. The results of these trials indicate that `nPointProbCrossover` sometimes results in earlier convergence than `nPointCrossover`. In all trials, both operators produced almost equivalent objective

scores. The pseudocode of `nPointProbCrossover` and `nPointCrossover` appear in Appendix D.

6.3.3.2 Mutation Operator

As is the case with crossover operators, GALib also defines default mutation operators for each of the types it defines. However, we do not make use of these and define our own mutation operator. As described in Section 5.2.4.2, a gene from a chromosome is randomly selected through a random number generator, and its value is altered along with any successive genes that are affected by the change. The pseudocode in Appendix D illustrates the implementation details of the mutation operator.

6.4 User Inputs

Various inputs to our application are required. These inputs can be decomposed into primarily three subgroups: task information, test environment and input options. Table 2 outlines the categories in each of the subgroups. These inputs are all combined in an input file.

Subgroup	Categories
task information	i
	ρ_i
	C_i
	dependencies
	r_i
	$\min \alpha_i$
	$\max \alpha_i$

task information	d_i
	<i>triggered tasks</i>
	<i>end times</i>
test environment	time interval
	target task
input options	show timing diagram

Table 2: *Application inputs*

Hence, for any task, its task number, priority, estimated execution time and dependencies (if any) must be specified. The scheduling strategy we adopt (Section 6.1) determines our definition of task dependencies. Two tasks are dependent if they share a resource. Tasks may optionally define other tasks that they trigger. Furthermore, periodic tasks must additionally define the period of the task, while aperiodic tasks define the minimum task interarrival time, maximum task interarrival time (if it exists) and the task deadline. Periodic task can optionally define end times, hence defining limited periodic tasks that execute only for a defined time span. It is important to note that the ending time is relative from the starting time of the periodic task.

Two values describing the test environment are also required as input: target task and time interval. The target task identifies a particular task whose deadlines are observed. The target task is the one that the Real-Time Test application tries to find a sequence of inputs that will lead to a miss in one of its deadlines. Because target tasks tend to be critical ones, and because users know the time critical tasks in the application under test, selection of the target task is left to the user. Selection of the time interval for testing is also left to the user. As a guideline, perhaps the ideal time interval would be one based on the LCM of all task periods. Allowing users to specify the time interval results

in twofold advantages: it enables users to identify particular problem regions and test those individually, especially considering that users may know typical usage periods in the application under test, as well as avoids the drawback of using LCM as described in Section 5.2.2. The inputted time interval value determines the upper bound of the interval, as use of the Generalized Completion Time Theorem (Section 2.3.2) dictates a lower bound of zero.

Users can further specify in the input to our application whether they would like to view the resulting timing diagram of the input sequence resulting from our application. This allows users to visualize problem areas in the target task.

6.4.1 Input File Format

We define a set of keywords for all the categories identified in the previous section. These are presented in Table 3. Each keyword is followed by a number, indicating its value. The only exceptions are *sched*, which is not followed by a number, and *dependent* as well as *triggers*, which may both be followed by more than one number separated by white space.

Category	Keyword
i	task
p_i	priority
C_i	exec
dependencies	dependent
r_i	period
min_i	min
max_i	max
d_i	deadline
triggered tasks	triggers

end time	end
time interval	interval
target task	target
show timing diagram	sched

Table 3: *Input keywords*

A few restrictions exist on the ordering of the keywords. *Interval* and *target* must appear before any task information as chromosome and task initializations depend on both these values. Furthermore, all information pertaining to one task must be listed before information relating to a different task. The ordering of the task information is left up to the user, although the *task* keyword indicating the task number should appear first. The optional *sched* can appear anywhere within the input file. These restrictions are encoded into our application's error handling mechanism. Users are prompted if any problems with the input file are detected. A sample input file appears in Appendix A.

6.4.2 Command-line Options

We provide our application with two optional command-line switches: *ifile* and *ofile* (both switches are followed by the file name). The former option designates the input file name where all input categories described in the previous section are listed. The latter option designates the output file name where the outputs of our application (Section 6.5) appear. We provide default values for the input and output files, if they are not specified: *gaInput.txt* and *score.txt* respectively

6.5 Application Outputs

The output file of our application houses the solution to the task set of inputs, a permissible error range for the execution time estimates above which deadline misses

occur, along with a number of statistics. A sample output file is presented in Appendix B. The output file first duplicates the input file for user's convenience. The aperiodic arrival times found for the target task are then listed in the following format (A_i, a_{ij}) . The objective score of the corresponding chromosome is also given. Differences between e_{ij} and d_{ij} are listed for the target task, with positive values indicating deadline misses for the corresponding task execution. If the *sched* option is enabled in the input file, the timing diagram corresponding to the solution is included in the output file. Error ranges for execution estimates are given in the forms of percentages. For example, execution estimates are safe under 2% means that if the estimates are accurate within 2%, no deadline misses will occur. The new differences e_{ij} and d_{ij} are listed for the target task. If the *sched* option is enabled, a timing diagram with the new estimates is included in the output file. Then, the best scores objective for every 10 generations of the GA are given. These show the rate of convergence of the solution. Finally, statistics and optimization customization are echoed onto the output file. These include: number of selections since initialization, number of crossovers since initialization, number of mutations since initialization, number of replacements since initialization, number of chromosome evaluations since initialization, number of population evaluations since initialization, maximum score since initialization, minimum score since initialization, average of all scores, average of maximum scores, average of minimum scores, mean score in initial population, maximum score in initial population, minimum score in initial population, standard deviation of initial population, mean score in final population, maximum and minimum scores in final population, standard deviation of final population, number of

generations, convergence percentage, crossover probability, mutation probability, and population size.

Chapter 7 – Case Studies

We report results for a number of case studies, using the principles and prototype tool described in the previous chapter. 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 7.2 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 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.

7.1 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 . GCTT proves that these three tasks are schedulable (details are provided in Appendix C). 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 GCTT, 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 t_3 . 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.

7.2 Schedulable Tasks Missing Deadlines Case Study

Consider the three tasks: t_1 ($r_{T1} = 3$, $C_{T1} = 1$, t , $p_{T1} = 32$, dependent on t_3), t_2 ($min_{T2} = 8$, $C_{T2} = 3$, $p_{T2} = 31$), t_3 ($r_{T3} = 9$, $C_{T3} = 2$, $p_{T3} = 30$, 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 Appendix G 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.

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.

7.3 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.

Task	Priority	Period (ms)	Execution Time (ms)
Timer Interrupt	100	1	0.051
Weapon Release	98	200	3
Radar Tracking Filter	84	25	2
RWR Contact Management	72	25	5
Data Bus Poll Device	68	40	1
Weapon Aiming	64	50	3
Radar Target Update	60	50	5
Navigation Update	56	59	8
Display Graphic	40	80	9
Display Hook Update	36	80	2
Tracking Target Update	32	100	5
Weapon Protocol	28	Aperiodic	1
Navigation Steering Commands	24	200	3
Display Stores Update	20	200	1
Display Keypad	16	200	1
Display Status Update	12	200	3
BET E Status Update	8	1000	1
Navigation Status	4	1000	1

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 time¹⁰ 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 *triggers* (to allow tasks to trigger other tasks) and *end* (to define tasks with limited time spans). For the purpose of this case study, we assume that any input received

¹⁰ Release calculation is not provided by the authors.

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. Recall from Section 6.1 that we require integer execution time estimates. 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 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 RTTT 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.

t	a_{weapon protocol,j}	j where e_{t,j}>d_t	Value of Deadline Miss (ms)	Execution Estimation Error
Weapon Release	4887, 11919, 12122, 14749, 14990	-	-	22.2%
Weapon Release Subtask	9999, 14066, 14630, 14947	-	-	22.2%
Radar Tracking Filter	5, 207, 512, 870, 1197, 1587, 1827, 2213, 2444, 2660, 3024, 3249, 3495, 3852, 4159, 4373, 4658, 4870, 5070, 5393, 5593, 5801, 6065, 6304, 6553, 6784, 7040, 7249, 7453, 7669, 7900, 8110, 8488, 8859, 9075, 9459, 9668, 9915, 10183, 10472, 11083, 11422, 11811, 12219, 12441, 12789	-	-	1155.5%
RWR Contact Management	39, 321, 522, 738, 1090, 1341, 1574, 1945, 2335, 2541, 2754, 3143, 3346, 3660, 3989, 4300, 4544, 4841, 5050, 5302, 5671, 5875, 6091, 6440, 6803, 7025, 7260, 7518, 7760, 8114, 8360, 8724, 9021, 9333, 9593, 9915, 10132, 10350, 10570, 10938, 11308, 11552, 11754, 11966, 12221, 12535, 12769, 13089, 13372, 13572, 13856, 14090	147, 548	3, 9	0%
Data Bus Poll Device	90, 299, 601, 892, 1093, 1295, 1687, 1896, 2100, 2415, 2802, 3048, 3261, 3476, 3684, 3893, 4094, 4294, 4509, 4730, 4931, 5180, 5440, 5797, 6015, 6244, 6460, 6669, 6884, 7093, 7308, 7525, 7790, 8014, 8343, 8737, 9099, 9330, 9533, 9850, 10051, 10263, 10556, 10756, 10969, 11174, 11887, 12693, 13274, 14981	-	-	211.1%
Weapon Aiming	94, 302, 508, 748, 1123, 1396, 1626, 1882, 2098, 2371, 2580, 2888, 3094, 3294, 3676, 3945, 4145, 4347, 4733, 5021, 5386, 5774, 6120, 6382, 6594, 6799, 7011, 7389, 7596, 7922, 8179, 8387, 8592, 9135, 9650, 10153,	-	-	22.2%

	10354, 10571, 10780, 11077, 11362, 12008, 12210, 12436, 12825, 13092, 13316, 13550			
Radar Target Update	107, 433, 653, 858, 1241, 1460, 1674, 2099, 2343, 2678, 2897, 3115, 3319, 3581, 3917, 4118, 4318, 4525, 4726, 5046, 5284, 5557, 5795, 6001, 6301, 6506, 6707, 6960, 7269, 7536, 7888, 8139, 8380, 8599, 8858, 9233, 9537, 9874, 10078, 10282, 10546, 10758, 11013, 11324, 11530, 11792, 11999, 12202, 12572, 12826, 13026, 13257	174, 214, 218, 267	17, 16, 10, 9	0%
Navigation Update	157, 368, 579, 781, 998, 1377, 1589, 1816, 2043, 2317, 2593, 2829, 3036, 3376, 3595, 3861, 4108, 4408, 4633, 5003, 5214, 5419, 5624, 5858, 6125, 6333, 6536, 6787, 7174, 7408, 7611, 7833, 8199, 8588, 8859, 9142, 9356, 9615, 9850, 10060, 10309, 10550, 10814, 11111, 11415, 11669, 11877, 12097, 12325, 12552, 12784, 12995, 13208, 13419, 13666, 13914, 14119, 14336, 14536, 14739	98, 101, 118, 132, 135, 190, 245	1, 29, 23, 2, 28, 27, 32	0%

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.

Target Task	Run Number	Objective Function Value	Largest Value of $e_{T,j} - d_T$
Weapon Release	1-10	0	-197
Weapon Release Subtask	1,2,4,6-10	0	-997
	3	0	-994
	5	0	-995
Radar Tracking Filter	1	0.191298	-3
	2	0.116635	-4
	3	64.002	6
	4	2.00288	1
	5	0.0889019	-4
	6	0.0427746	-5
	7	0.559302	-1
	8	0.0685323	-4
	9	18.0067	4
	10	0.130897	-3
RWR Contact Management	1	520.035	9
	2	112.96	5
	3	32.094	5
	4	552.665	8
	5	2049.11	11
	6	16664.2	14
	7	8640.07	13
	8	136.587	7
	9	403.03	8
	10	162.134	7
Data Bus Poll Device	1	0.0371494	-5
	2	0.562528	-1
	3	0.257822	-2
	4	2.01564	1
	5	0.0629904	-4

	6	0.039091	-5
	7	0.00172283	-12
	8	0.000150355	-14
	9	16.0001	4
	10	0.500017	-1
Weapon Aiming	1	0.0201419	-6
	2	0.00586035	-8
	3	0.00595128	-8
	4	0.126953	-3
	5	0.000214935	-13
	6	0.0195313	-6
	7	0.000489237	-11
	8	0.157288	-3
	9	0.0312508	-5
	10	0.00394637	-8
Radar Target Update	1	198144	17
	2	1058300	20
	3	1048580	20
	4	139265	17
	5	140288	17
	6	262146	18
	7	590081	19
	8	163840	17
	9	81921.2	16
	10	24578.3	14
Navigation Update	1	5242880000	32
	2	6442450000	32
	3	18387800000	34
	4	17179900000	34
	5	67148000	26
	6	138513000000	37

	7	72351700	26
	8	146029000000	37
	9	18547300000	34
	10	309240000000	38

Table 6: *RTTT results of 10 runs*

Chapter 8 – 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 Chapter 7 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 thesis 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 thesis 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 thesis 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 7.2. 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

- [1] Weyuker, Elaine J. and Filippou I. Vokolos. "Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study". *IEEE Transactions on Software Engineering*, Vol. 26, No. 12, pp. 1147-1156, December 2000.
- [2] Zhang, Jian and S.C.Cheung. "Automated Test Case Generation for the Stress Testing of Multimedia Systems". *Software - Practice and Experience*, Vol. 32, pp. 1411-1435, 2002.
- [3] Yang, Cheer-Sun D. and Lori L. Pollock. "Towards a Structural Load Testing Tool". *ACM SIGSOFT Software Engineering Notes*, Vol. 21, Issue 3, May 1996.
- [4] Liu, Jane W. S. *Real-Time Systems*. New Jersey: Prentice Hall, 2000.
- [5] Gomaa, Hassan. *Designing Concurrent, Distributed, and Real-time Applications With UML*. Boston MA: Addison-Wesley, 2000.
- [6] Sprunt, Brinkley. "Aperiodic Task Scheduling for Real-Time Systems". PhD Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1990.
- [7] Sha, Lui and John B. Goodenough. "Real-time Scheduling Theory and Ada". Technical Report. CMU/SEI-89-TR-014, ESD-TR-89-022, *Software Engineering Institute*, April 1989.
- [8] Klein, Mark H. et. al. *A Practitioner's Handbook for Real-Time Analysis – Guide to Rate Monotonic Analysis for Real-Time Systems*. Software Engineering Institute, Carnegie Mellon University. Boston: Kluwer Academic Publishers, 1993.
- [9] Cooling, Jim. *Software Engineering for Real-Time Systems*. England: Addison-Wesley, 2003.
- [10] Liu, C. L. and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, pp 46-61, January 1973.
- [11] Balarin, Felice et. al. "Scheduling for Embedded Real-Time Systems". *IEEE Design and Test of Computers*, Vol. 15, Issue 1, pp. 71-82, January-March 1998.
- [12] Hong, Tzung-Pei et. al. "Simultaneously Applying Multiple Mutation Operators in Genetic Algorithms". *Journal of Heuristics*, 6:439-455, 2000.
- [13] Hou, Edwin S. H. et. al. "A Genetic Algorithm for Multiprocessor Scheduling". *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5. No. 2, February 1994.

- [14] Wall, Matthew. "GAlib: A C++ Library of Genetic Algorithm Components". Version 2.4, Document Revision B. Massachusetts Institute of Technology, August 1996.
Available at <http://lancet.mit.edu/ga/dist/galibdoc.pdf>
- [15] Locke, C. Douglass, David R. Vogel and Thomas J. Mesler. "Building a Predictable Avionics Platform in Ada: A Case Study". *Proceedings of the IEEE 12th Real Time Systems Symposium*. Pp. 181-189, December 1991.
- [16] Bäck, Thomas. "Self-Adaptation in Genetic Algorithms". *Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. Pp. 263-271. MIT Press, 1992.
- [17] Smith, J. E. and T. C. Fogarty. "Adaptively Parameterized Evolutionary Systems: Self Adaptive Recombination and Mutation in a Genetic Algorithm". *Parallel Problem Solving From Nature 4*. Eds. Voigt, Ebeling, Rechenberg and Schwefel. Pp. 441-450, 1996.
- [18] Haupt, Randy L. and Sue Ellen Haupt. *Practical Genetic Algorithms*. New York: John Wiley & Sons, 1998.
- [19] Goldberg, David E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston: Addison-Wesley, 1989.
- [20] Jacob, Christian trans. *Illustrating Evolutionary Computation with Mathematica*. San Francisco: Morgan Kaufmann Publishers, 2001.
- [21] Eiben, A.E., P-E. Raué, and Zs. Ruttkay. "Solving Constraint Satisfaction Problems Using Genetic Algorithms". *Proceedings of the IEEE World Conference on Evolutionary Computation*, 1994.
Available at <ftp://ftp.cs.vu.nl/pub/papers/ai/WCCI.EibenRaueRuttkay.ps.Z>.
- [22] Pawlowsky, Marc Andrew. *Practical Handbook of Genetic Algorithms Applications Volume I*. Ed. Lance Chambers. Boca Raton: CRC Press, 1995.
- [23] Atallah, Mikhail J. ed. *Algorithms and Theory of Computation Handbook*. Boca Raton: CRC Press, 1999.
- [24] Chinneck, John W. *Practical Optimization: A Gentle Introduction*. Draft. Canada, 2003.
Available at <http://www.sce.carleton.ca/faculty/chinneck/po.html>
- [25] Fukunaga, Alex S., Andre Stechert and Steve Chien. "Towards a Self-Configuring Optimization System for Spacecraft Design". *International Symposium on Artificial Intelligence, Robotics and Automations in Space*. Tokyo, Japan. July 1997.

- [26] Battiti, Roberto and Giampietro Tecchiolli. "The Reactive Tabu Search". *ORSA Journal on Computing*. Vol. 6 No. 2, pp. 126-140, 1994.
Available at: <http://rtm.science.unitn.it/~battiti/archive/reactive-tabu-search.ps.gz>
- [27] Lahtinen, Jussi, et. al. "Empirical Comparison of Stochastic Algorithms". *Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications*. Alander, Jarmo T. Ed. pp. 45-60. Finland, 1996.
Available at: <ftp://ftp.uwasa.fi/cs/2NWGA/Lahtinen.ps.Z>
- [28] Manikas, Theodore W. and James T. Cain. "A Comparison of Heuristics for Telecommunications Traffic Routing". *Modern Heuristic Search Methods*. Rayward-Smith et. al Ed. John Wiley & Sons, 1996.
- [29] Mahfouz, S. "Design Optimization of Structural Steel Work". PhD Thesis, Department of Civil and Environmental Engineering, University of Bradford, 1999.
- [30] Mahfoud, Samir W. and David E. Goldberg. "Parallel Recombinative Simulated Annealing: A Genetic Algorithm". *Parallel Computing*. Vol. 21 No. 1, January 1995.
- [31] Forrest, Stephanie and Melanie Mitchell. "What Makes a Problem Hard for a Genetic Algorithm? Some Anomalous Results and their Explanation". *Machine Learning*. Vol. 13, 1993.

Appendix A – Sample Input File

interval 10

target 1

task 1 period 3 priority 32 exec 1 dependent 3

task 2 min 8 deadline 100 priority 31 exec 3

task 3 period 9 priority 30 exec 2 dependent 1

sched

Appendix B – Sample Output File

Input:

interval 10

target 1

task 1period 3 priority 32 exec 1 dependent 3

task 2min 8deadline 100 priority 31 exec 3

task 3period 9 priority 30 exec 2 dependent 1

sched

The worst aperiodic arrival times found are (objective score is 3):

(2, -1) (2, 2)

Differences between target task execution end time and deadline time ($e_{i,j} - d_{i,j}$):

Task execution 1: -2

Task execution 2: 1

Task execution 3: -1

Task execution 4: -2

Timing Diagram of tasks:

	1	2	3
0	#		
1			#
2		#	
3		#	
4		#	
5			#
6	#		
7	#		
8			
9	#		

Execution estimates are safe under 0%

Differences between target task execution end time and deadline time ($e_{i,j} - d_{i,j}$) with estimate error percentage:

Task execution 1: -2

Task execution 2: 1

Task execution 3: -1

Task execution 4: -2

Timing Diagram of tasks with estimate error percentage:

	1	2	3
0	#		
1			#
2		#	
3		#	
4		#	
5			#
6	#		
7	#		
8			
9	#		

Statistics:

Best of generation scores:

Generation Score

0	3
10	3
20	3
30	3
40	3
50	3
60	3
70	3
80	3
90	3
100	3

100 # current generation

1 # current convergence

4000 # number of selections since initialization

2814 # number of crossovers since initialization

4082 # number of mutations since initialization

4000 # number of replacements since initialization

4080 # number of genome evaluations since initialization

101 # number of population evaluations since initialization

3 # maximum score since initialization

1 # minimum score since initialization

2.83631 # average of all scores ('on-line' performance)

3 # average of maximum scores ('off-line' performance)

2.6425 # average of minimum scores ('off-line' performance)

1.24375 # mean score in initial population

3 # maximum score in initial population

```
1 # minimum score in initial population
0.596635 # standard deviation of initial population
-1 # diversity of initial population (0=identical,-1=unset)

3 # mean score in current population
3 # maximum score in current population
3 # minimum score in current population
0 # standard deviation of current population
-1 # diversity of current population (0=identical,-1=unset)

20 # how far back to look for convergence
10 # how often to record scores
10 # how often to write scores to file
gaBestScores.txt # name of file to which scores are written

minimaxi 1
number_of_generations 100
generations_to_convergence 20
convergence_percentage 0.99
crossover_probability 0.7
mutation_probability 0.015468
population_size 80
score_frequency 10
flush_frequency 10
record_diversity 0
score_filename gaBestScores.txt
select_scores 2
number_of_best 1
replacement_percentage 0.5
replacement_number 40
```

Appendix C – Proof of Schedulability of Execution

Estimates Case Study

- t1: $r_{T1} = 255$, $C_{T1} = 200$, $\rho_{T1} = 32$
- t2: $\min_{T2} = 240$, $C_{T2} = 20$, $\rho_{T2} = 31$
- t3: $r_{T3} = 250$, $C_{T3} = 20$, $\rho_{T3} = 30$

Modelling the aperiodic task as a periodic task with period equivalent to the minimum interarrival time and using the Generalized Completion Time Theorem (Section 2.3.2) to determine whether the tasks are schedulable, we obtain:

1. For t1

$$i=1, k=1, l_{k=1} = 1.. \left\lfloor \frac{255}{255} \right\rfloor = 1$$

$$\frac{200}{255} \leq 1$$

$$200 \leq 255$$

TRUE

Hence, t1 will meet its first deadline.

2. For t2

$$i=2, k=1,2, l_{k=1} = 1.. \left\lfloor \frac{240}{255} \right\rfloor = 1..0$$

The values for l here are unexpected as they span 1..0. Evidently, the GCTT does not take priority inversion into consideration for the value of l . Nevertheless, we can ignore this and assume that t2 can not complete its execution within the

period of t_1 . Instead, we check for completion within its own period. Hence, using

$k=2$, we obtain:

$$l_{k=2} = 1.. \left\lfloor \frac{240}{240} \right\rfloor = 1$$

$$200\left(\frac{1}{240}\right) \left\lceil \frac{240}{255} \right\rceil + \frac{20}{240} \leq 1$$

$$200 + 20 \leq 240$$

$$220 \leq 240$$

TRUE

Hence, t_2 will meet its deadline before its first deadline.

3. For t_3

$$i=3, k=1,2,3, l_{k=1} = 1.. \left\lfloor \frac{250}{255} \right\rfloor = 1..0$$

Again, we are faced with the same problem where the values of l span an improper range. Hence, we consider values that are dependent on other values of k

$$l_{k=2} = 1.. \left\lfloor \frac{250}{240} \right\rfloor = 1$$

$$l_{k=2} = 1$$

$$200\left(\frac{1}{240}\right) \left\lceil \frac{240}{255} \right\rceil + 20\left(\frac{1}{240}\right) \left\lceil \frac{240}{240} \right\rceil + \frac{20}{240} \leq 1$$

$$200 + 20 + 20 \leq 240$$

$$240 \leq 240$$

TRUE

Hence, t_3 will meet its deadline before the first deadline of t_2 .

From 1, 2 and 3 above, this group of tasks is thus schedulable according to the Generalized Completion Time Theorem.

Appendix D – Enhanced Metamodel

Constraints:

```

Chromosome::getTaskExecsHead(taskNumber): int
pre: self.Task.taskNumber->asSet->includes(taskNumber)
post: result= self.Gene.select(self.Gene.taskNumber==taskNumber)->at(1)

Chromosome::move(gene, positions, enum(left, right))
pre: positions>=0 && positions<self.gene->size - gene.location
and
self.gene.taskNumber->asSet->includes(gene)
post: gene.location = gene.location@pre + positions

Chromosome::generateNewArrivalTime(prevArrival, taskNumber): int
pre: prevArrival >= -1 and self.Task.taskNumber->asSet->includes(taskNumber)
post: result >= self.Task->atTaskNumber.minInterarrivalTime
and
result <= self.Task->atTaskNumber.maxInterarrivalTime
    
```

Figure 28: Enhanced Metamodel

The `GA1DArrayGenome<Gene>` object shown in Figure 28 is defined in `GAlib`. `Chromosome` inherits from `GA1DArrayGenome<Gene>`, which defines all array operations. The two operations, `nPointCrossover` and `nPointProbCrossover` are the implementations of the two flavours of crossover operators described in Section 5.2.4.1. T depicts the testing time interval while t designates the target task. `Gene` represents a gene of the chromosome. It contains information about the arrival time and task number of the corresponding gene. It further contains information pertaining to its location within the chromosome. In the `Task` object, which is the superclass containing both periodic and aperiodic tasks, k denotes the maximum number of times the corresponding task can execute within the given time frame `Chromosome.T`. `Dependencies` contain the list of tasks that are dependent on the given task. Similarly, `triggers` contain the list of tasks the given task triggers, or initiates.

Appendix E - Pseudocode

```
procedure Chromosome::Initializer(Chromosome chromosome)
var: currentGene: int //current gene in chromosome
      Ai: int          //task number
      j: int           //execution number
      minRange: int    //minimum range
      maxRange: int    //maximum range
      newArrivalTime: int //randomly generated arrival time
function: random      //random number generator

//Initializes the chromosome with random arrival times that conform to
//the minimum and maximum interarrival times
begin
  currentGene = 1
  for (Ai=1; Ai <= chromosome.aperiodicTask->size; Ai++) do //for all
                                                    //genes
    for (j=1; j<= chromosome.aperiodicTask->at(Ai).k; j++) do //for all
                                                    //task executions
      //set task number
      chromosome.gene->append(new Gene())
      chromosome.gene->at(currentGene).setTaskNumber(Ai)
      if (j==1) then
        minRange = 0
        maxRange = chromosome.aperiodicTask->at(Ai).getMaxTime()
        if (maxRange==0) then
          maxRange = T
        endif
      else //(j==1)
        minRange =
          chromosome.gene->at(currentGene-1).getArrivalTime() +
          chromosome.aperiodicTask->at(Ai).getMinTime()
        maxRange =
          chromosome.gene->at(currentGene-1).getArrivalTime() +
          chromosome.aperiodicTask->at(Ai).getMaxTime()
        if (chromosome.aperiodicTask->at(Ai).getMaxTime() == 0)
          then
            maxRange =
              chromosome.gene->at(currentGene-1).getArrivalTime() + T
          endif
        endif //(j==1)
      //set arrival time
      newArrivalTime = random (minRange, maxRange)
      chromosome.gene->at(currentGene).setArrivalTime(newArrivalTime)
      if (newArrivalTime >= T) then
        chromosome.gene->at(currentGene).setArrivalTime(-1)
```

```
        move(currentGene,j-1,left)
    endif
    currentGene = currentGene + 1
end // for all task executions
end // for all genes
end
```



```

procedure Chromosome::move(int gene, int positions, enum {left, right}
                                direction)

var: currentGene: int //current gene in chromosome
      i: int           //gene number
function: swap        //switches location of two genes

//move current gene the number of times specified by positions in
//the direction specified by direction
begin
  currentGene=gene;
  if (direction == left) then
    for (i=1; i<=positions; ++i,--currentGene) do
      swap(currentGene, currentGene-1)
    end
  else
    for (i=1; i<=positions; ++i,++currentGene) do
      swap(currentGene, currentGene+1)
    end
  endif
end

```

```

function Chromosome::nPointProbCrossover(in parent1: Chromosome,
                                         in parent2: Chromosome,
                                         inout child1: Chromosome,
                                         inout child2: Chromosome): int

var: mom,           //copy of parent1
      dad,           //copy of parent2
      sis,           //reference to child1
      bro,           //reference to child2
      nAperiodicTaskHead, //location of head of aperiodic task
      nAperiodicTaskTail: int //location of tail of aperiodic task
function: GAFlipCoin //coin flip simulator
          copy         //copies chromosomes

//This crossover operator defines n points for crossover, where
//n = number of tasks -1. For each set of task executions, a coin flip
//determines whether or not the two parent executions are switched in
//the children.
begin
//initialization
    nAperiodicTaskHead =0

//copy parents and children
    mom = parent1
    dad = parent2
    sis = child1
    bro = child2
    copy(sis,mom)
    copy(bro,dad)

//loop on tasks in one parent
    for (taskCounter= 0; taskCounter <
        mom.aperiodicTask->size; taskCounter++) do
        if (GAFlipCoin(0.5)) then
            //flip returns true; modify tail
            nAperiodicTaskTail = nAperiodicTaskHead +
                mom.aperiodicTask->
                    at(taskCounter).getK()-1;

            //swap all executions of task
            for (execCounter= nAperiodicTaskHead;
                execCounter<= nAperiodicTaskTail; execCounter++) do
                swap(execCounter, bro, execCounter)
            end
        endif //GAFlipCoint(0.5)
        //set head to next aperiodic task's head
        nAperiodicTaskHead = nAperiodicTaskHead +
            mom.aperiodicTask->at(taskCounter).getK();
    end //loop on tasks in one parent
//return the number of children created

```

```
    return 2;  
end
```

```
procedure swap(geneLocation1: int, otherChromosome: Chromosome,  
               geneLocation2: int)  
//the operation swaps the gene at location geneLocation1 in the current  
//chromosome with the gene at location geneLocation2 in the  
//otherChromosome.
```

```

function Chromosome::nPointCrossover(in parent1: Chromosome,
                                     in parent2: Chromosome,
                                     inout child1: Chromosome,
                                     inout child2: Chromosome): int

var: mom,           //copy of parent1
      dad,           //copy of parent2
      sis,           //reference to child1
      bro,           //reference to child2
      nAperiodicTaskHead, //location of head of aperiodic task
      nAperiodicTaskTail: int //location of tail of aperiodic task
      bSwitch: bool      //indicates whether to switch fragments
function: swap        //swaps genes in chromosome
           copy         //copies chromosomes
//This crossover operator defines n points for crossover, where
//n = number of tasks -1. For each set of task executions, alternate
//pairs from the parents are switched in the children.
begin
    //initialization
    nAperiodicTaskHead =0;
    bSwitch = false;

    //copy parents and children
    mom = parent1;
    dad = parent2;
    sis = child1;
    bro = child2;
    copy(sis,mom)
    copy(bro,dad)

    //loop on tasks in one parent
    for (taskCounter= 0; taskCounter <
        mom.aperiodicTask->size; taskCounter++) do
        if (bSwitch) then
            //swap all executions of task
            for (execCounter= nAperiodicTaskHead;
                execCounter<= nAperiodicTaskTail; execCounter++) do
                swap(execCounter, bro, execCounter)
            end
            bSwitch = false;
        else bSwitch = true;
        endif
        //set head to next aperiodic task's head
        nAperiodicTaskHead = nAperiodicTaskHead +
            mom.aperiodicTask->at(taskCounter).getK();
    end //loop on tasks in one parent
    //return the number of children created
    return 2;
end

```

```

function Chromosome::Mutator(inout genome: Chromosome, float pmut): int
var:  nGeneLocation,           //location of selected gene
       nTaskHead,              //location of first task execution of
                                //chosen task
       nTaskTail,              //location of tail task execution
       numMoves,               //total number of altered genes
       nTaskNumber,            //task number of selected gene
       nPrevArrivalTime,       //arrival time of previous gene
       nMin,                   //minimum arrival time of gene
       nMax: int               //maximum arrival time of gene
       selectedGene,           //selected gene
       currentGene,            //holds current gene
       previousGene,           //holds previous gene
       tailGene: Gene          //tail gene of task
function: random              //random number generator
       generateNewArrivalTime  //generates new arrival time based on
                                //minimum and maximum interarrival
                                //times
       move                    //moves genes in chromosome

//This mutation operator randomly chooses a gene then randomly
//generates a new arrival time for it as well as any subsequent genes
//affected by it to ensure maximum and minimum interarrival
//time constraints are upheld. If the choosen gene has arrival
//time = -1, the random generation depends on the last gene of the
//task. If the newly generated number is still -1, a new gene is
//chosen.
begin
  //repeat as long as no gene is affected
  for (;;) do
    nGeneLocation = random(1,genome->size);
    selectedGene = genome.gene->at(nGeneLocation);

    nTaskNumber = selectedGene.getTaskNumber();
    nTaskHead = genome.getTaskExecsHead(nTaskNumber);
    nTaskTail = nTaskHead +
                genome.aperiodicTask->at(nTaskNumber).getK()-1;
    tailGene = genome.gene->at(nTaskTail);

    nMin = genome.aperiodicTask->at(nTaskNumber).getMinA();
    nMax = genome.aperiodicTask->at(nTaskNumber).getMaxA();

    numMoves=0;

    //check for 3 cases:
    //case 1: chosen gene not the first execution, last execution, or
    //one with arrival time -1
    //case 2: chosen gene has arrival time -1
    //case 3: chosen gene is the first execution of the task

```

```

//case 2: -1 arrival time genes
if (selectedGene.getArrivalTime()==-1) then
    if (genome.insertGene(nGeneLocation,numMoves)) then
        return numMoves;
    endif
//cases 1 and 3
else //(selectedGene.getArrivalTime()==-1)
    //select new arrival times for selected gene and affected
    //executions of task

    //case 1: if task head not chosen for mutation, set previous
    //gene and arrival time to previous gene
    if (counter!=nTaskHead)then
        previousGene = genome.gene->at(nGeneLocation-1)
        nPrevArrivalTime = previousGene.getArrivalTime()
    else //(counter!=nTaskHead)
        //case 3: first task execution chosen for mutation
        nPrevArrivalTime = -1
    endif //(counter!=nTaskHead)

    //set new arrival time for chosen gene
    currentGene.setArrivalTime(
        genome.generateNewArrivalTime(nPrevArrivalTime,nTaskNumber))
    if (currentGene.getArrivalTime()>=genome.T)then
        currentGene.setArrivalTime(-1)
        genome.move(currentGene.getLocation(),
            currentGene.getLocation()-nTaskHead,LEFT)
        numMoves= currentGene-nTaskHead+1
    endif

    numMoves++
    for (counter=nGeneLocation+1; counter<=nTaskTail; counter++) do
        currentGene = genome.gene->at(counter)
        nPrevArrivalTime=genome.gene->at(counter-1).getArrivalTime()
        if (nPrevArrivalTime == -1) then
            tempArrivalTime = 0
        else
            tempArrivalTime = nPrevArrivalTime
        endif

        if ((currentGene.getArrivalTime() >= tempArrivalTime + nMin)
            &&
            (currentGene.getArrivalTime() <= tempArrivalTime + nMax))
            then
                break;
            endif

        numMoves++

```

```

currentGene.setArrivalTime(
    genome.generateNewArrivalTime(nPrevArrivalTime,nTaskNumber))

if (currentGene.nArrivalTime>=genome.T)then
    currentGene.setArrivalTime(-1)
    genome.move(counter,counter-nTaskHead,LEFT)
    numMoves= counter-nTaskHead+1
endif
end // for (counter=nGeneLocation+1; counter<=nTaskTail; counter++)
//check last gene
if (tailGene.getArrivalTime()==-1) then
    if (genome.T - tailGene.getArrivalTime() > nMax ) then
        int nNewMoves = 0
        genome.insertGene(0,nNewMoves)
        numMoves += nNewMoves
    endif
endif
endif //(selectedGene.getArrivalTime()==-1)
return numMoves
end // for (;;)
end

```



```

function Chromosome::getTaskExecsHead(int nTaskNumber): int
//retrieve head of nTaskNumber
begin
    for (taskCounter=1;taskCounter<=self.gene->size;taskCounter++) do
        if (self.gene->at(taskCounter).getTaskNumber()==nTaskNumber) then
            return self.gene->at(taskCounter).getLocation()
        endif
    return -1
end

```

```

function Chromosome:: generateNewArrivalTime(int prevArrivalTime,
                                             int taskNumber): int
var: nPrevArrivalTime: int           //copy of previous arrival time
      lowRange,                       //low range for random generator
      highRange: int                 //high range for random generator
function: random                    //random number generator
//generate new random number given previous arrival time and task
//number
begin
  nPrevArrivalTime = prevArrivalTime
  if (nPrevArrivalTime== -1) then
    lowRange = 0
    highRange = self.task->at(taskNumber).getMaxA()
  else
    lowRange = nPrevArrivalTime + self.task->at(taskNumber).getMinA()
    highRange = nPrevArrivalTime + self.task->at(taskNumber).getMaxA()
  endif
  return random (lowRange,highRange)
end

```

```

function Chromosome::insertGene(int nGeneLocation, inout moves:int)
                                : bool

var: selectedGene: Gene        //selected gene
      tailGene: Gene            //last gene of task
      task: Task                //task information
      nTaskLocation,            //task location
      nTaskHead,                //last gene of task
      numMoves,                 //total number of altered genes
      nextGeneArrival,          //arrival time of next gene
      previousGeneArrival,      //arrival time of previous gene
      nTaskHead: int            //first gene of task

function:
      random                    //random number generator

//insert -1 arrival time gene into the earliest available time slot
//such that maximum and minimum time constraints are met. Moves is an
//out parameter that keeps track of the number of moves made to
//the chromosome.
begin
  //insert -1 arrival time gene into the earliest available time slot
  //such that maximum and minimum time constraints are met
  selectedGene = self.gene->at(nGeneLocation)
  nTaskLocation = selectedGene.getLocation()
  nTaskHead = getTaskExecsHead(selectedGene.getTaskNumber())
  task = self.task->at(selectedGene.getTaskNumber())
  nTaskTail = nTaskHead + task.getK()-1
  tailGene = self.gene->at(nTaskTail)

  // find first non negative arrival time
  for (nNumMoves = nGeneLocation+1;
      self.gene->at(nNumMoves).getArrivalTime() == -1; nNumMoves++) do
    ;
  end

  if (nNumMoves > nTaskTail) then
    //whole genome filled with -1
    selectedGene.setArrivalTime(
      generateNewArrivalTime(tailGene.getArrivalTime(),
        selectedGene.getTaskNumber()))
    moves = 1
    move(nGeneLocation, nTaskTail - nGeneLocation, RIGHT)
    return true
  endif

  nextGeneArrival = self.gene->at(nNumMoves).getArrivalTime()

  //can insert before first non negative arrival time
  if (nextGeneArrival > task.getMinA()) then
    selectedGene.setArrivalTime(random(0,
      nextGeneArrival - task.getMinA()))
    move(nGeneLocation, (nNumMoves-1) - nGeneLocation, RIGHT)
    moves = 1
    return true

  // if max interarrival < 2 minimum interarrival, can only insert
  //gene at end
  else // (nextGeneArrival > task.getMinA())
    if (task.getMaxA() < 2 * task.getMinA()) then
      selectedGene.setArrivalTime(
        generateNewArrivalTime(tailGene.getArrivalTime(),
          selectedGene.getTaskNumber()))
      if (selectedGene.getArrivalTime() >= self.T) then

```

```

        selectedGene.setArrivalTime(-1)
        return false
    else //(selectedGene.getArrivalTime())>=self.T)
        moves = nTaskTail-nNumMoves+1
        move(nGeneLocation,nTaskTail-nGeneLocation,RIGHT)
        return true
    endif //(selectedGene.getArrivalTime())>=self.T)
endif //(task.getMaxA()<2*task.getMinA())
//try inserting between genes
else //(nextGeneArrival>task.getMinA())
    for (count=nNumMoves+1; count<=nTaskTail; count++) do
        previousGeneArrival = nextGeneArrival
        nextGeneArrival = self.gene->at(count).getArrivalTime()
        if ((nextGeneArrival-previousGeneArrival)>=
            (2* task.getMinA())) then
            selectedGene.setArrivalTime(
                random(previousGeneArrival + task.getMinA(),
                    nextGeneArrival-task.getMinA()))
            moves = count-nNumMoves+1
            move(nGeneLocation,count-nGeneLocation-1,RIGHT)
            return true
        endif
    end //for
endif //(nextGeneArrival>task.getMinA())
return false
end //insertGene

```

```

function Chromosome::Evaluator(Chromosome genome): float
var: nSum: float          //objective score
      vDiff: collection    //collection of differences of execution end
                          //times - deadlines for each task execution
function: pow             //raises a number to a power
//objective function that schedules all tasks and keeps track of the
//merit value of each chromosome
begin
  self.schedule()
  nSum = 0.0
  for (i = 1;i<=self.gene->size;i++) do
    nSum += pow(2,self.vDiff->at(i));
  end
  return nSum
end

```

```

procedure Chromosome::schedule()
var:
    triggerSet,                //set of triggered tasks
    runningSet: set<int>        //set of currently running tasks

    struct DepNode: public Gene
        nRemainingExec,        //remaining execution time
        nPriority,              //priority
        nDeadline:int           //relative deadline
        bLastPeriodicExec: bool //indicator of whether or not last
                                //periodic execution
    endStruct

    aperiodicGene,              //aperiodic task exec
    ready,                      //task exec ready to execute
    periodicGene: DepNode       //periodic task exec
    blockedSet,                 //set of blocked tasks
    taskExecSet: set<DepNode>   //set of tasks executions ordered
                                //by increasing arrival times
    readyExecs: priority_queue<DepNode> //queue of ready task
                                // executions sorted by priority
    vDepInterval: bag<int> (nTimeInterval)
                                //collection of time slots
    vDiff: bag<int>             //collection of differences of end
                                //times and deadline times for task
                                //executions
//schedules task executions with dependencies, triggers, and limited
//periods, as well as updates difference vector, vDiff, for target task
begin
    //add aperiodic tasks to task exec set
    for (i = 1; i<=self.gene->size; i++) do
        //include only task executions with arrival times
        currentGene = self.gene->at(i)
        if (currentGene.getArrivalTime() != -1) then
            aperiodicGene = currentGene
            aperiodicGene.nRemainingExec =
                self.task->at(currentGene.getTaskNumber()).getExecTime()
            aperiodicGene.nDeadline =
                self.task->at(currentGene.getTaskNumber()).getDeadline()
            aperiodicGene.bLastPeriodicExec = false

            taskExecSet.insert(aperiodicGene)
        endif
    end

    //add periodic tasks to task exec set
    for (i=1; i<=self.periodicTask->size; i++) do
        currentGene = self.task->at(i)
        //if task triggered by another task, skip it
        if (currentGene.getTriggeredBy() != -1) then
            continue
        endif

        periodicGene.setTaskNumber(currentGene.getTaskNumber())
        periodicGene.nRemainingExec = currentGene.getExecTime()
        periodicGene.bLastPeriodicExec = false
        for (j = 0; j<currentGene.getK(); j++) do
            periodicGene.setArrivalTime(j*currentGene.getPeriod())
            periodicGene.nDeadline = currentGene.getPeriod()
            if (j+1==currentGene.getK()) then
                periodicGene.bLastPeriodicExec = true
            endif
            taskExecSet.insert(periodicGene)
        end //for (j = 0; j<currentGene.getK(); j++)

```

```

end //for (i=1;i<=self.periodicTask->size;i++)

//loop on time interval and allocate time slots for task executions
for (timeSlot=0;timeSlot<self.T;timeSlot++) do
    //identify schedulable task executions for current time slot
    while (taskExecSet->notEmpty()) do
        if (taskExecSet->at(1)->getArrivalTime() > timeSlot) then
            break
        endif
        readyExecs.push(execIt)
        taskExecSet.erase(execIt)
    end
    // select the task to schedule at this time slot
    while (readyExecs->notEmpty()) do
        ready = readyExecs.top()

        // check dependencies
        set<int> setDep =
            self.task->at(ready.nTaskNumber).getDependencies()

        if (setDep->intersection(runningSet)->notEmpty()) then
            blockedSet.insert(ready)
            readyExecs.pop()
            continue
        endif

        //assign time slot
        vDepInterval->at(timeslot) = ready.nTaskNumber
        ready.nRemainingExec--

        //task execution complete
        if (ready.nRemainingExec!=0) then
            runningSet.insert(ready.nTaskNumber)
        else
            //update difference bag if target task
            if (ready.nTaskNumber== self.t) then
                vDiff.insert(
                    timeSlot-(ready.nArrivalTime + ready.nDeadline - 1))
            endif

            //add triggered task executions
            //periodic and aperiodic tasks can only trigger periodic tasks
            triggerSet= self.task->at(ready.nTaskNumber).getTriggers()

            for (triggIt = triggerSet->iterate(true)) do
                DepNode restrictedPeriodicGene

                if (self.periodicTask->at(triggIt)) then
                    PeriodicTask task = self.periodicTask->at(triggIt)
                    int period = task.getPeriod()

                    restrictedPeriodicGene.nTaskNumber = task.getTaskNum()
                    restrictedPeriodicGene.nRemainingExec = task.getExecTime()
                    restrictedPeriodicGene.nPriority = task.getPriority()

                    int jExecStart = timeSlot
                    int jExecEnd = timeslot + (period*task.getK())
                    if (task.getEndLimit()!=-1) then
                        jExecEnd = timeSlot+ task.getEndLimit()
                    endif

                    int jExecOrigEnd = jExecEnd

```

```

        if (jExecEnd>nTimeInterval)
            jExecEnd = self.T-1

        if (ready.bLastPeriodicExec==false) then
            if (self.periodicTask) then
                continue
            endif
        endif
        for (j = jExecStart; j<=jExecEnd;j+=period) do
            restrictedPeriodicGene.nArrivalTime=j+1
            restrictedPeriodicGene.nDeadline = period
            //if last periodic execution
            if ((j+period>jExecOrigEnd)&&
                (self.periodicTask->at(triggerIt))) then
                restrictedPeriodicGene.bLastPeriodicExec = true
            else
                restrictedPeriodicGene.bLastPeriodicExec = false
            endif
            taskExecSet.insert(restrictedPeriodicGene)
        end //for (j = jExecStart; j<=jExecEnd;j+=period)
        end // (self.periodicTask->at(triggerIt))
    end //for (triggerIt = triggerSet->iterate(true))

    //remove from running set
    runningSet.erase(ready.nTaskNumber)

    //remove from ready list
    int nFinishedTask = ready.nTaskNumber
    readyExecs.pop()

    //unblock blocked task executions
    bag<int> del
    for (it = blockedSet->iterate(true)) do
        //get task dependencies of task in blocked set
        set<int> setDep = self.task->at(it).getDependencies()
        if (setDep->exists(nFinishedTask) then
            //add to ready list
            readyExecs.insert(it)
            //add to blocked execs to be deleted
            del.insert(it)
        endif
    end

    //remove from blocked set
    for (delCount = 0; delCount<del->size;delCount++) do
        blockedSet.erase(delCount)
    end //(ready.nRemainingExec!=0)
    break;
end //while
end //for (i = 0;i<self.gene->size;i++)
end

```


Appendix F – GAlib Patch

```
diff --unified --recursive --new-file ga/GAAllele.C newGA/GAAllele.C
--- ga/GAAllele.C   Thu Nov 11 13:56:04 1999
+++ newGA/GAAllele.C   Fri Oct 10 14:34:32 2003
@@ -132,8 +132,8 @@
// When we link to another allele set, we point our core to that one. Be sure
// that we have a core. If not, just point. If so, trash as needed.
template <class T> void
-GAAlleleSet<T>::link(GAAlleleSet<T>& set){
- if(&set != this){
+GAAlleleSet<T>::link(GAAlleleSet<T>& theSet){
+ if(&theSet != this){
+   if(core != 0){
+     core->cnt -= 1;
+     if(core->cnt == 0) delete core;
diff --unified --recursive --new-file ga/GAArray.h newGA/GAArray.h
--- ga/GAArray.h   Thu Nov 11 13:56:04 1999
+++ newGA/GAArray.h   Thu May  8 03:03:44 2003
@@ -31,7 +31,7 @@
class GAArray {
public:
  GAArray(unsigned int s) : sz(s), a(sz ? new T[sz] : 0)
-   {for(unsigned int i=0; i<sz; i++) a[i] = (T)0; }
+   {for(unsigned int i=0; i<sz; i++) a[i] = T(); }
  GAArray(const GAArray<T> & orig){sz=0; a=(T *)0; copy(orig);}
  GAArray<T> & operator=(const GAArray<T> & orig){copy(orig); return *this;}
  GAArray<T> & operator=(const T array []) // no err checks!
```

Appendix G – Proof of Schedulable Tasks Missing

Deadlines Example

- 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

1.) For t1

$$i=1, k=1, l_{k=1} = 1.. \left\lfloor \frac{3}{3} \right\rfloor = 1$$

$$\frac{1}{3} + \frac{2}{3} \leq 1$$

$$1 + 2 \leq 3$$

$$3 \leq 3$$

TRUE

Hence, t1 will meet its first deadline.

2.) For t2

$$i=2, k=1,2, l_{k=1} = 1.. \left\lfloor \frac{8}{3} \right\rfloor = 1,2$$

$$l_{k=1} = 1$$

$$1\left(\frac{1}{3}\right)\left\lceil \frac{3}{3} \right\rceil + \frac{3}{3} \leq 1$$

$$1 + 3 \leq 3$$

$$4 \leq 3$$

FALSE

$$l_{k=1} = 2$$

$$1\left(\frac{1}{6}\right)\left\lceil\frac{6}{3}\right\rceil + \frac{3}{6} \leq 1$$

$$2 + 3 \leq 6$$

$$5 \leq 6$$

TRUE

Hence, t2 will meet its deadline before the second deadline of t1.

3.) For t3

$$i=3, k=1,2,3, l_{k=1} = 1.. \left\lfloor \frac{9}{3} \right\rfloor = 1,2,3$$

$$l_{k=1} = 1$$

$$1\left(\frac{1}{3}\right)\left\lceil\frac{3}{3}\right\rceil + 3\left(\frac{1}{3}\right)\left\lceil\frac{3}{8}\right\rceil + \frac{2}{3} \leq 1$$

$$1 + 3 + 2 \leq 3$$

$$6 \leq 3$$

FALSE

$$l_{k=1} = 2$$

$$1\left(\frac{1}{6}\right)\left\lceil\frac{6}{3}\right\rceil + 3\left(\frac{1}{6}\right)\left\lceil\frac{6}{8}\right\rceil + \frac{2}{6} \leq 1$$

$$2 + 3 + 2 \leq 6$$

$$7 \leq 6$$

FALSE

$$l_{k=1} = 3$$

$$1\left(\frac{1}{9}\right)\left\lceil\frac{9}{3}\right\rceil + 3\left(\frac{1}{9}\right)\left\lceil\frac{9}{8}\right\rceil + \frac{2}{9} \leq 1$$

$$3 + 6 + 2 \leq 9$$

$$11 \leq 9$$

FALSE

$$l_{k=2} = 1.. \left\lfloor \frac{9}{8} \right\rfloor = 1$$

$$l_{k=2} = 1$$

$$1\left(\frac{1}{8}\right)\left\lceil\frac{8}{3}\right\rceil + 3\left(\frac{1}{8}\right)\left\lceil\frac{8}{8}\right\rceil + \frac{2}{8} \leq 1$$

$$3 + 3 + 2 \leq 8$$

$$8 \leq 8$$

TRUE

Hence, t3 will meet its deadline before the first deadline of t2.

From 1, 2 and 3 above, this group of tasks is thus schedulable according to the Generalized Completion Time Theorem.

Appendix H – Alternative Representations

Alternative Chromosome Representation

Alternatively, the chromosome can be represented with a repetition of integers, $a_{A1,1} a_{A2,1} a_{A3,1} * a_{A1,2} a_{A2,2} a_{A3,2} * a_{A1,3} a_{A2,3} a_{A3,3}$. Here, task numbers are eliminated and arrival times of all tasks are contiguous. The arrival times for all tasks' first executions appear first, followed by the arrival times for the next executions, etc... For example, the sequence 8 9 50 * 25 20 55 * 30 30 100 depicts three aperiodic tasks, t1, t2 and t3 with arrival times occurring at time units 8, 25 and 30 for t1; 9, 20 and 30 for t2; 50, 55 and 100 for t3. Hence, the task number associated with an arrival time can be inferred from its position. The arrival times of a task are ordered according to increasing order. For example, this sequence would be considered illegal: 8 9 50 * 25 20 55 * 10 30 100. The second and third executions of t1 are not in increasing order. The exception to the increasing constraint is if the task arrival time is -1. In this case, all subsequent task executions must also be -1. Hence, another illegal sequence would be: 8 9 50 * -1 20 55 * 10 30 100. Here, the third execution of t1 should be -1 instead of 10. There are many drawbacks to this representation. It is confusing as some information is not explicit and must be inferred. Furthermore, it introduces far more constraints than the representation we define.

Alternative Chromosome Length

The length of the chromosome can be defined by the greatest value of period and minimum interarrival times of all tasks. While plausible, this generates an unnecessarily

lengthy chromosome, most of which will be unused. In Figure 22, the greatest minimum interarrival time of all three tasks is 200. This would thus be indicative of a chromosome of length 200. However, as equation 6 reveals, a chromosome of length five is sufficient.

Alternative Objective Function

Recall that our search for an objective function is confined by a number of criteria that define some desirable properties:

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 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.

Based on these three criteria, we investigate alternative objective functions. Recall that the objective function we adopt sums the differences of all the executions of the target

task. Alternatively, the objective function can adopt a minimum value technique by selecting the minimum target task execution difference, i.e. $f(c_i) = \max_{j=1..k} (2^{e_{i,j}-d_{i,j}})$. While this function meets criteria one and three, it fails to meet the second criteria. Let us consider an example where the target task has three executions and two chromosomes have target task execution differences $(d_{i,j} - e_{i,j})$ $c1 = 2, 10, 1$ and $c2 = 15, 20, 1$. Using the minimum value technique, we obtain $f(c1) = \max (2^{-2}, 2^{-10}, 2^{-1}) = 0.5$ and $f(c2) = \max (2^{-15}, 2^{-20}, 2^{-1}) = 0.5$. Hence, both chromosomes would be considered equivalent. Although the two chromosomes are just one time unit away from the third execution's missing its deadline, $c1$ additionally sees another execution that is close to meeting its deadline: the first execution. Because execution times are mere estimates and are subject to inaccuracy (Section 2.2), $c1$ would be more likely to result in a deadline miss than $c2$. This technique would thus fall short. However, summing the differences yields $f(c1) = 2^{-2} + 2^{-10} + 2^{-1} = 0.7509$ and $f(c2) = 2^{-15} + 2^{-20} + 2^{-1} = 0.5000$. Because this is a maximization function, $c1$ is deemed fitter than $c2$, as was intuitively concluded.

Conversely, consider a linear objective function. In other words, the rewarding and penalization of chromosomes is performed in a linear fashion: $f(c) = \sum_{j=1}^{k_t} T - (d_{i,j} - e_{i,j})$. Here, smaller differences yield larger values than larger differences. However, the sizes of rewards and penalties are not adequate. Hence, criteria one and two are met, while the third is not. Over the whole chromosome, small differences can be overshadowed and even overtaken by larger differences. This can best be illustrated with an example. Assume we are evaluating the fitness of two chromosomes, $c1$ and $c2$ for a target task with three executions and $T = 200$. The values

for $d_{i,j} - e_{i,j}$ for $c1 = 1, 10, 10$ and $c2 = 2, 50, 1$. Using the equation, $f(c1) = 199 + 190 + 190 = 579$ and $f(c2) = 198 + 150 + 199 = 547$. From the evaluation of $f(c1)$ and $f(c2)$, $c1$ has a larger value, hence implying that it has smaller differences and is thus fitter. However, $c2$ should be fitter since it has two values that are close to missing their deadlines. Like $c1$, it has an execution that is just one away from breaking its deadline. It also has another execution that is two away from breaking its deadline. This is the result obtained using Equation 8, whereby $f(c1) \approx 0.502$ and $f(c2) \approx 0.750$.

It is interesting to note that the two functions defined above represent the two extremes of the second criteria. In the former function, only the worst task execution is considered while in the latter, all task executions are taken into consideration. It is important to note that the objective function we define in our approach is more of a compromise between these two extremes. It considers all task executions, yet rewards bad task executions highly. In a sense, it gives more weight to the worst task execution.

Intuitively, the fitness of chromosomes could have been defined using the inverse of the difference of execution end time and deadline: $f(c) = \sum_{j=1}^{k_t} \frac{1}{d_{i,j} - e_{i,j}}$. Here, smaller differences yield higher values for $f(c)$ than larger ones. All three criteria are not met. Furthermore, using the inverse of the difference is undefined when $d_j = e_{i,j}$. In addition, if $d_{i,j} < e_{i,j}$, the inverse yields a negative value which is interpreted as a penalty. This should not be the case, as an execution that has exceeded its deadline should be rewarded. A solution to the first problem would be the addition of a constant, x , i.e. $f(c) = \sum_{j=1}^{k_t} \frac{1}{d_{i,j} - e_{i,j} + x}$.

The denominator of the fraction in this case cannot be undefined. Nevertheless, this solution does not solve the second problem. The value of the constant x would need to be

defined at one point. However, because the task set is not set, the chosen value of x might not be appropriate, still yielding a negative value. Taking the negative of the equation would not solve the second problem either. Hence, $f(c) = \sum_{j=1}^{k_i} \frac{1}{e_{i,j} - d_{i,j} - x}$ would reward smaller differences and penalize larger ones. The same problem arises when $d_{i,j} < e_{i,j}$. Depending on the value of x , chromosomes with this property can be rewarded, penalized, or undefined.