Universidade Federal da Bahia
Escola Politécnica / Instituto de Matemática
Programa de Pós-Graduação em Mecatrônica

FLÁVIA MARISTELA SANTOS NASCIMENTO

A SIMULATION-BASED FAULT RESILIENCE ANALYSIS FOR REAL-TIME SYSTEMS

DISSERTAÇÃO DE MESTRADO

Salvador
2009

FLÁVIA MARISTELA SANTOS NASCIMENTO

A SIMULATION-BASED FAULT RESILIENCE ANALYSIS FOR REAL-TIME SYSTEMS

Dissertação apresentada ao Programa de Pós-Graduação em Mecatrônica da Escola Politécnica e do Instituto de Matemática, Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre.

Orientador: *Prof. Dr. George Marconi de Araújo Lima*
Co-orientadora: *Profa. Dra. Verônica Maria Cadena Lima*

Salvador
2009

Sistemas de Bibliotecas - UFBA

Nascimento, Flávia Maristela Santos.
A simulation-based fault resiliense analysis for real-time systems / Flávia Maristela Santos Nascimento. - 2009.
104 f.

Orientador: Prof. Dr. George Marconi Lima.
Co-orientadora: Profa. Dra. Verônica Maria Cadena Lima.
Dissertação (mestrado) - Universidade Federal da Bahia, Instituto de Matemática e Escola Politécnica, Salvador, 2009.

1. Programação em tempo real. 2. Controle em tempo real. 3. Tolerancia a falha (Computação). 4. Distribuição (Probabilidades). I. Lima, George Marconi. II. Lima, Verônica Maria Cadena. III. Universidade Federal da Bahia. Instituto de Matemática. IV. Universidade Federal da Bahia. Escola Politécnica. V. Título.

CDD - 004.33
CDU - 681.3.014

TERMO DE APROVAÇÃO

Título da Dissertação A SIMULATION-BASED FAULT RESILIENCE ANALYSIS FOR REAL-TIME SYSTEMS

Estudante FLÁVIA MARISTELA SANTOS NASCIMENTO

Dissertação aprovada como requisito parcial para a obtenção do grau de Mestre em Mecatrônica, Universidade Federal da Bahia – UFBA, pela seguinte banca examinadora:

[illegible]

Prof. Dr. George Marconi de Araújo Lima (Orientador)
Ph.D. em Ciência da Computação, University of York, Inglaterra
Professor da Universidade Federal da Bahia

[illegible]

Prof. Dr. Luciano Porto Barreto (Examinador PPGM)
Doutor em Informática, Universidade de Rennes, França
Professor da Universidade Federal da Bahia

[illegible]

Prof. Dr. Rômulo Silva de Oliveira (Examinador Externo)
Doutor em Engenharia Elétrica, Universidade Federal de Santa Catarina, Brasil
Professor da Universidade Federal de Santa Catarina

02 de Outubro de 2009

A Romildo, pelo incentivo, paciência e amor constante.
A Avanir, Edgar e Flávio, pedras fundamentais.

Aprender é a única coisa de que a mente nunca se cansa, nunca tem medo e nunca se arrepende.

—LEONARDO DA VINCI

AGRADECIMENTOS

Inicialmente, agradeço a Deus, por ter me guiado para superar todas as adversidades que aconteceram nesta fase. Agradeço a minha mãe Avanir, porque há muito tempo ela é a maior responsável por todas minhas conquistas. Para você, mãe, minha reverência. A meu pai, Edgar, simplesmente por estar vivo. A Flávio, meu querido irmão, sempre atento, sempre zeloso. À Romildo, um presente de Deus, pelos momentos indescritíveis de apoio, incentivo, cuidado, paciência e amor. Pelas madrugadas em que insistia em ficar acordado para me fazer companhia, pelos códigos que me ajudou a depurar, pelos mimos e surpresas, que vou guardar com especial carinho.

Agradeço aos amigos, que apesar da distância sempre se fizeram presentes, fosse para me ouvir ou para não me deixar esquecer de minhas obrigações e objetivos: Fabiano Almeida, Danuza Neiva, Marcos Camada, Ecivaldo Matos e Talmai Oliveira. Ao meu amigo e padrinho, Pablo Vieira e àquele sempre disposto a tirar um sorriso a qualquer custo, Flávio Campos.

A meu orientador, George Lima, pela forma com que conduziu o trabalho, sempre encontrando algum tempo para me ouvir, mesmo diante de tantas atribuições. Sua paciência, empenho e dedicação foram fundamentais para a realização deste trabalho. Sempre vou guardar comigo seus ensinamentos, principalmente o cuidado em todas as revisões de texto e artigos e a constante preocupação com a qualidade.

À Verônica Lima, pela paciência em me mostrar os viés da Estatística e pelo cuidado especial nos detalhes finais.

A todos os professores e funcionários do PPGM pelo apoio na realização deste trabalho.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

RESUMO

Sistemas de tempo real tem sido amplamente utilizados no contexto de sistemas mecatrônicos uma vez que, para controlar entidades do mundo real, é necessário considerar tanto seus requisitos lógicos quanto os temporais. Em tais sistemas, mecanismos para prover tolerância a falhas devem ser implementados já que falhas podem implicar em perdas consideréveis. Por exemplo, um erro em um sistema de controle de vôo pode incorrer em perda de vidas humanas.

Várias abordagens de escalonamento com tolerância a falhas para sistemas de tempo real foram derivadas. Entrento, a maioria delas restringe o modelo de sistema e/ou falhas de modo particular, ou estão fortemente acopladas ao modelo de recuperação do sistema ou a política de escalonamento. Além disso, não existe uma métrica formal que permita comparar as abordagens existentes do ponto de vista da resiliência a falhas. O objetivo principal deste trabalho é preencher esta lacuna, fornecendo uma métrica de resiliência a falhas para sistemas de tempo real, que seja o mais independente possível dos modelos do sistema e/ou de falhas. Para tanto, uma análise baseada em simulação foi desenvolvida para calcular a resiliência de todas as tarefas de um sistema, através da simulação de intervalos de tempo específicos. Em seguida, técnicas de inferência estatística são utilizadas para inferir a resiliência do sistema. Os resultados mostraram que a métrica desenvolvida pode ser utilizada para comparar, por exemplo, duas políticas de escalonamento para sistemas de tempo real sob a ótica de resiliência a falhas, o que demonstra que a abordagem desenvolvida é razoavelmente independente do modelo de sistema.

Palavras-chave: Sistemas de Tempo Real, Escalonamento, Tolerância a Falhas

ABSTRACT

Mechatronics systems are characterized by their ability of integrating components and systems to control real world entities. To do so, real-time systems have increasingly been used, since controlling real world objects requires considering both their logical and timing constraints. In this context, fault tolerance mechanisms play an important role since faults in such a system may imply in considerably losses. For example, an error in a flight control system may incur in loss of human life.

Several fault-tolerant approaches for real-time systems have been derived. Most of them restrict the system and fault model in a particular way or are strictly linked to a specific recovery model or scheduling policy. Moreover, to the best of our knowledge, there is no systematic metric, which enables an effective comparison among the existing fault-tolerant approaches. The main goal of this work is to fill this gap by providing a fault resilience comparison metric for different fault-tolerant scheduling approaches. To do so, a simulation-based framework was derived to compute resilience values while simulating specific time windows. Such values are then used to infer system resilience based on statistical analysis techniques. Results have shown that the derived metric can be used to compare, for example, two scheduling policies for real-time systems from fault resilience viewpoint, which demonstrates that our approach is reasonably independent of the system model.

Keywords: Real-Time System, Scheduling, Fault Tolerance

CHAPTER 1

INTRODUCTION

Mechatronics, as sub-area of automation, is gaining prominence due to its ability to integrate components and systems (via hardware or software), in order to make cost-saving products [36, 27]. The main components of a mechatronic system is shown in Figure 1.1.

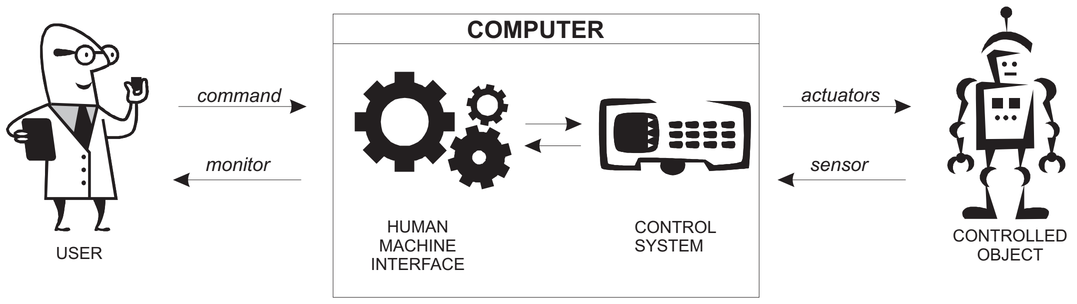


Figure 1.1. General scheme of a mechatronic system

The *user* is the entity that needs/wishes to manipulate a *controlled object*, which can be an industrial plant or a robot, for example. The controlled object is a real-world entity that has logical requirements, related to its functional correctness, and timing requirements, usually defined as the maximum time to perform its operations. Controlled objects are usually manipulated by the user through a *Human Machine Interface* (HMI), which aims at monitoring the operation and commanding this object. In most applications the human machine interface interacts with controlled objects through a *control system*, which is responsible for managing the entire operation flow of controlled objects. *Sensors* and *actuators* are used by control systems to perform operations in controlled objects. The former are devices responsible for informing the current state of the controlled objects to control systems. The later represents the mechanisms that enable the control system to interfere in controlled objects evolution [39].

Real-time systems have increasingly been used in mechatronics since these applications necessarily need to consider both logical and timing requirements of controlled objects. It is interesting to notice that there is an important difference between general purpose systems and real-time systems. The former focus on assuring logical correctness and minimizing task average response times. For real-time systems correctness depends not only on the logical results but also on the time at which results are produced [50].

Real-time systems are usually defined based on a set of tasks, where each task represents an execution unit. Usually, each task has its own timing constraint. For example, the task that runs a control algorithm should respond within a maximum predetermined time interval. Thus, it is necessary to establish an execution order for tasks (or its operations) considering its timing requirements. Generating such a sequence, also called execution scale or schedule, is an attribution of the scheduling mechanism, a key component for ensuring timeliness.

Usually, the schedule is generated based on heuristics, called scheduling policies, that define which tasks will be executed at each time instant. Once the scheduling policy is chosen, it is necessary to verify if there is a possibility that any task violates its timing restriction. Thus, tasks execution order, generated by the scheduler, must be validated to ensure timeliness. This activity is performed by the schedulability analysis, which will evaluate the system temporal behavior, checking if all tasks will meet their timing requirements. If this is true, the system is said to be schedulable according to the considered scheduling policy.

In the context of real-time applications, we must also consider the possibility of error occurrence. In some cases, upon the detection of an error, re-executing the faulty task can be satisfactory for providing fault tolerance. In others, recovery tasks need to be executed to ensure that the system is put in a safe state. These fault tolerance techniques are characterized by using time redundancy and can be effective for several real-time systems [24, 17, 32].

Indeed, on the assumption that all applications potentially fail [56], we must evaluate if faults can prevent tasks from meeting their deadlines, particularly with regard to applications in which time violation may lead to serious consequences. Actually, when errors take place, the time available for fault tolerance is smaller than in scenarios where there are no errors. Thus, not only the scheduler should cover such possibilities, but also the analysis techniques must be suitable to consider errors consequences.

Several fault-tolerant scheduling approaches have been derived. In a nutshell, they artificially assume a given worst-case scenario for error occurrences and then adapt schedulability analysis techniques accordingly. Also, the derived approaches are strictly linked to specific system and/or fault model. For example, some approaches consider that errors take place periodically [17, 8]. Others, fix a maximum number of errors per system task [4, 32].

Although these techniques are important, because they give designers some sort of timeliness assessment, they do not provide a measurement of system fault resilience. Indeed, it is interesting to have some measurement of fault resilience for real-time systems, which can be independent of the system and the assumed error pattern, issue not yet satisfactorily addressed by the research community.

The main goal of this work is to fill this gap by providing a metric which allows the comparison between different real-time systems from fault resilience point of view. To do so, a simulation-based framework is derived to compute fault resilience values while simulating specific time windows. Also, some statistical analysis is carried out to infer the fault resilience of a given system based on a sample of such values. Results can then be used to subsidize a quantitative comparison among real-time systems. Some preliminar results has been presented recently [42, 34, 43] showing that the derived metric can be used, for example, to compare two mostly used scheduling policies for real-time systems.

The remainder of this document is organized as follows. Chapter 2 presents basic concepts on real-time systems in terms of task set, scheduling policies and feasibility analysis approaches. Also, this chapter describes some scheduling approaches for dealing with error occurrences. Chapter 3 presents the simulation-based approach, developed to compute a fault resilience metric for real-time systems, and some important definitions and properties. The main components in this environment are the *scenario generator*, which helps to determine the simulation time interval, and the *error generator* which effectively computes the fault resilience for each system task. These components are described in Chapters 4 and 5, respectively. The statistical analysis of data is presented in Chapter 6. Finally, conclusions and future work are presented in Chapter 7.

CHAPTER 2

REAL TIME SYSTEMS AND FAULT TOLERANCE

Real-time systems are usually described as a set of tasks, with logical and timing constraints, that need to be executed in some specific order so that such constraints are not violated. Tasks execution order is determined by scheduling policies and timeliness requirements are assessed by schedulability analysis techniques. When the system is subject to unexpected events such as errors, both scheduling policy and schedulability analysis must be adapted to consider fault tolerance aspects.

This chapter gives an overview of some concepts of a real-time system, the most common scheduling policies and feasibility analysis techniques and then shows how fault tolerance has been considered in such systems.

2.1 REAL-TIME SYSTEMS STRUCTURE

2.1.1 Tasks

Real-time systems are usually structured as a set of tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, where each task τ_i represents an execution unit. Tasks are characterized by having precedence relations, communication requirements and other attributes. For each task τ_i, some of the most important attributes are computation time, period and deadlines [22]. Such attributes are shown in Figure 2.1.

When task τ_i is released for execution it spends at most C_i time units executing, which represents the worst-case execution time required by such a task. Also, each task $\tau_i \in \Gamma$ must be executed within a specific time interval, which is called deadline and is denoted by D_i. Deadlines represent task timing constraints. Notice that τ_i can be activated several times (three times in Figure 2.1) and each of these activations is called

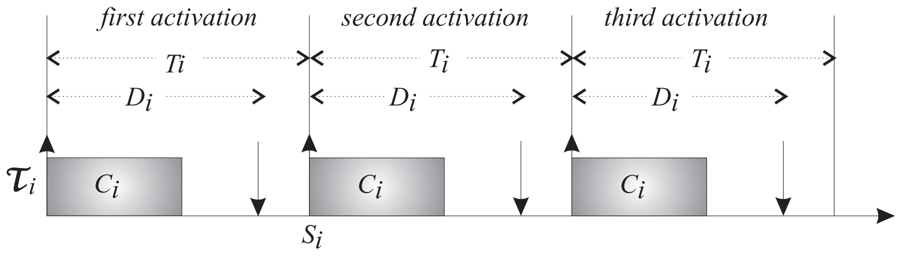


Figure 2.1. Illustration of a periodic task τ_i

a *job*. Each job is released at a time instant, illustrated by up arrows in the figure.

In general, tasks in real-time systems may have different criticality levels with respect to their time constraints. "Critical" real-time systems may potentially cause serious consequences if task deadlines are not met [22, 50]. They are also called *hard* real-time systems and tasks are said to have *hard* deadlines. This is the case, of the robots in an automobile production line or a flight control system, for example. On the other hand, for "non-critical" real-time systems, violating task deadlines does not necessarily imply in environmental risks or loss of human life. Such applications are also known as *soft* real-time systems and tasks are said to have *soft* deadlines. This is the case of bank processing systems and multimedia applications.

Also, observe in Figure 2.1 that τ_i is released at each T_i time units, which represents its activation periodicity. Depending on its activation occurrence tasks can be classified as *periodic*, when they are time-triggered on a regular basis as illustrated in Figure 2.1 or *aperiodic*, when they have random activation time instant as shown in Figure 2.2.

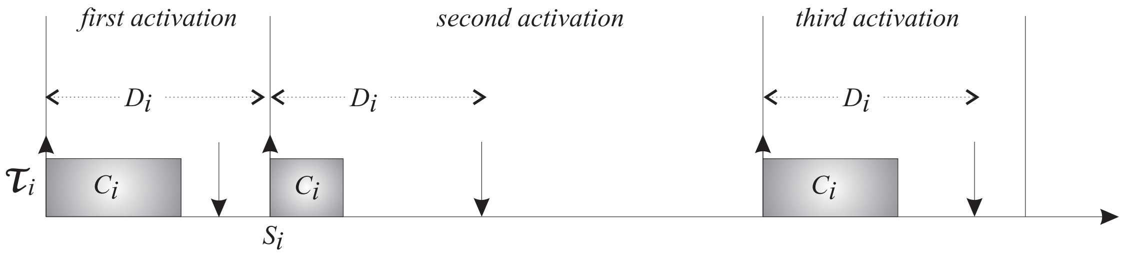


Figure 2.2. Illustration of an aperiodic task τ_i

It is also possible to describe *sporadic* tasks, which are aperiodic but have known minimum interval between two consecutive activations. Consequently, the maximum load a sporadic task τ_i may impose to the system is when it arrives periodically, since T_i represents the lower bound between its successive activations [48].

Based on the described attributes, tasks are represented by four *n-tuples*, which give their periods, deadlines, worst-case execution times and recovery times, respectively denoted as $\mathbf{T} = (T_1, \dots, T_n)$, $\mathbf{D} = (D_1, \dots, D_n)$, $\mathbf{C} = (C_1, \dots, C_n)$ and $\bar{\mathbf{C}} = (\bar{C}_1, \dots, \bar{C}_n)$.

Tasks attributes described above represent an important parameter for the scheduler, since based on them the tasks execution order is defined and timeliness assessment is carried out. The execution order for each task is determined by heuristics, called scheduling policies, which will be described in the following section.

2.1.2 Scheduling Policies

The schedule is defined by the *scheduling policy* and the component responsible for generating the schedule is the *scheduler*, which can be *preemptive*, when a task execution can be interrupted, and *non-preemptive*, otherwise [22]. A schedule that enables the execution of all tasks without violating their timing constraints, is named *feasible* [35].

The most popular scheduling policies are priority-oriented. Under such policies each task/job has a priority associated with it, which is responsible for establishing precedence relations among those that are ready to execute. Thus, the scheduler chooses to run that task/job ready for execution which has the highest priority. Although priority in real-time systems can be associated according to several criteria, for priority-oriented policies, the precedence between tasks is usually determined on the basis of its timing constraints, since this favors the implementation of better performance scheduling algorithms [22]. Nonetheless, it is still possible to map priorities according to other attributes such as degree of importance, for example [10].

Tasks priorities can be fixed, when it is established during the project (*off line*), or dynamic, when the priority is defined during runtime (*on line*) [22]. When task priorities are defined during runtime there is a gain in terms of flexibility and adaptability, what allows greater adequacy to situations that cannot be predicted, such as errors [37, 51]. In any case, the implementation of scheduling policies should be simple and efficient, allowing timeliness validation through schedulability analysis. The most popular scheduling policies are summarized in the following sections.

2.1.2.1 Fixed-Priority Scheduling Policies

Before presenting two of the most popular fixed-priority scheduling policies, consider Example 2.1, which is used as illustration.

Example 2.1 *Consider a periodic task set composed of two tasks* $\Gamma = \{\tau_1, \tau_2\}$. *Tasks attributes are represented by tuples such that* $\mathbf{C} = (3,3)$, $\mathbf{T} = (8,12)$ *and* $\mathbf{D} = (7,5)$ *represent their worst-case execution time, period and deadline respectively.*

Rate Monotonic (RM) is a fixed-priority scheduling policy according to which tasks priorities are assigned in inverse order of their periods. Since these periods are fixed per task, all jobs of a given task have the same priority. When a task deadline is equal to its period ($D_i = T_i$) and tasks are independent of each other, RM is considered an optimal algorithm within the class of fixed priority algorithms, since it is able to schedule any task set that can be scheduled by any other fixed priority scheduling policy [35, 6, 22]. Figure 2.3 shows the RM scheduling for Example 2.1. Down arrows represents task deadlines.

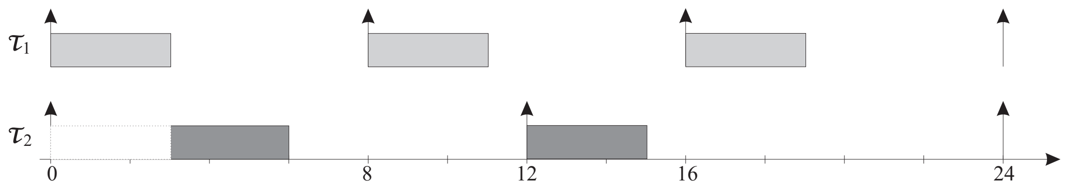


Figure 2.3. Illustration of a RM schedule for Example 2.1

Observe that both tasks are released together at time instant 0, however, since τ_1 has the highest priority it executes first and τ_2 waits until it finishes. This is represented by the dotted lines. Also, notice that the first job of τ_2 misses its deadline, which is not desired, in the general case. The scheduling for Γ is carried out during the hyperperiod h, which is the least common multiple of task periods. In this case, $h = \text{lcm}(T_1, T_2) = 24$.

As mentioned before, RM optimality considers that task periods are equal to their deadlines and this represents a restriction for some applications [22]. The *Deadline Monotonic* (DM) scheduling policy can be seen as an extension of RM. Actually, DM is another fixed priority scheduling policy in which the highest priority task is the one with lowest relative deadline. Figure 2.4 shows a DM scheduling for Example 2.1.

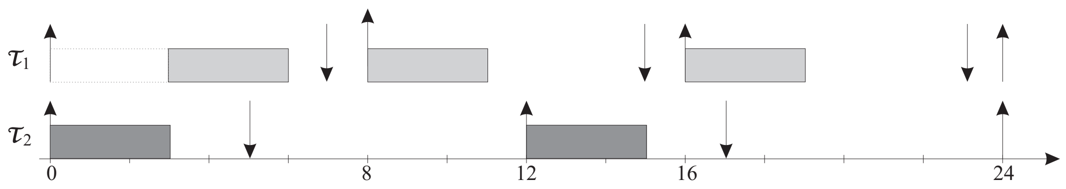


Figure 2.4. Illustration of a DM schedule for Example 2.1

When task deadlines are less than or equal to their period ($D_i \leq T_i$), Deadline Monotonic is considered to be an optimal algorithm among those fixed-priority scheduling policies [22, 6, 47].

2.1.2.2 Dynamic-Priority Scheduling Policies

One of the most usual scheduling policies based on dynamic priorities is known as *Earliest Deadline First* (EDF). According to EDF, jobs with the shortest deadline have the highest priority, and therefore are executed first. This means that different jobs from the same task may have different priorities. In the context of priority-oriented scheduling policies (fixed or dynamic), EDF is considered optimal for preemptive systems with one processor [35, 6, 47].

While fixed-priority policies assume constant deadlines for each job, EDF requires an extra computation to update such priorities. Furthermore, EDF presents a better performance than RM, when context switches are considered, since the number of preemptions made by a RM scheduler is greater than the one considering EDF [10]. To understand

EDF scheduling, consider the following example as illustration.

Example 2.2 *Consider a periodic task set composed of two tasks* $\Gamma = \{\tau_1, \tau_2\}$*. Tasks attributes are represented by tuples such that* $\mathbf{C} = (1, 4.5)$*,* $\mathbf{T} = (3, 8)$ *and* $\mathbf{D} = (2, 7)$ *represent their worst-case execution times, periods and deadlines respectively. Figure 2.5 shows an EDF schedule for* Γ*.*

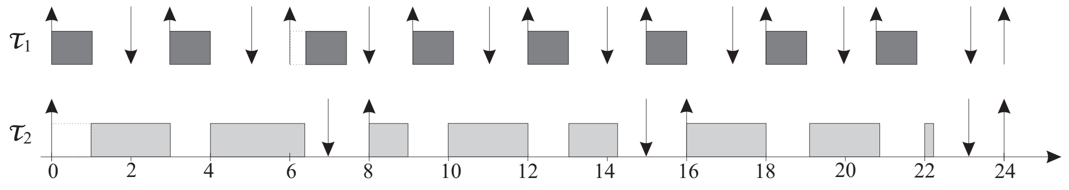


Figure 2.5. Illustration of a EDF schedule for Example 2.2

Observe that if RM was being considered, at time instant 6 τ_1 would preempt τ_2 since its period is smaller. However, when EDF is considered τ_2 is not preempted. The decision about which scheduling policy should be used for an application depends on its purpose. Each approach has advantages and disadvantages that should be considered in the context of the system model and the application criticality.

2.1.3 Schedulability Analysis

As mentioned before, schedulability analysis is a means of checking whether tasks meet their deadlines considering a given scheduling policy. Intuitively, it is possible to verify if all tasks in a given a periodic task set Γ will meet their deadlines simulating an execution scale for them during the hyperperiod. However, depending on the relation between task periods, the generated execution scale can be too large. Moreover, such method is static and not flexible since if a task needs to be added or removed from the task set, the hyperperiod value needs to be recalculated and execution scale has to be generated again.

It is worth mentioning that in some cases, because of the large number of tasks in the task set, it is difficult to make a simulation encompassing all possible combinations for tasks. In practice, schedulability analysis techniques look forward to defining system schedulability in terms of mathematical functions. Three kinds of feasibility analysis are described in literature [22, 37]:

- Sufficient tests: used to determine a schedulability upper bound. For values beyond this upper bound there are no schedulability guarantees;
- Necessary tests: used to determine unschedulability bounds, identifying unschedulable task sets;
- Exact tests: they are both sufficient and necessary. In other words, this kind of test can identify schedulable and unschedulable task sets, and consequently both schedulability and unschedulability bounds are known.

Figure 2.6 illustrates the tests described previously.

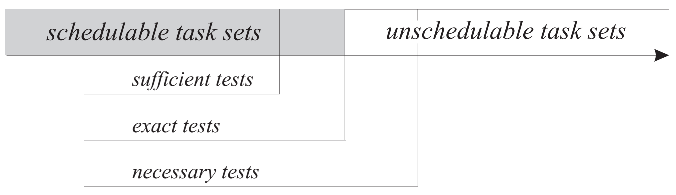


Figure 2.6. Schedulabity tests [22]

In general, two approaches to schedulability tests are common: Processor Utilization Analysis and Response Time Analysis. Such techniques are described in Sections 2.1.3.1 and 2.1.3.2, respectively.

2.1.3.1 Processor Utilization Analysis

Suppose a set of n independent, periodic tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$, scheduled by a uniprocessored system. Each task $\tau_i \in \Gamma$ is activated periodically and executes at most C_i time units. Thus, each task uses at most $\mathbf{U}_i = C_i/T_i$ of the processor total capacity. In other words, the processor maximum utilization factor to this task set is given by:

$$\mathbf{U} = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.1}$$

The schedulability analysis is given in terms of a simple mathematical function, taking into consideration the processor utilization factor. Considering a task set with n tasks scheduled by RM the system is said to be schedulable if [35]:

$$\mathbf{U} \leq n(2^{\frac{1}{n}} - 1) \tag{2.2}$$

Note that for large values of n this bound approaches to $\ln 2 \approx 69\%$. This implies that for systems with $U > 69\%$, it may not be possible to determine the system schedulability.

Hence, fixed-priority scheduling policies, this analysis is characterized as sufficient, since it is not able to identify all schedulable task sets. As illustration, consider that $\mathbf{D} = \mathbf{T}$ in Example 2.1. In this case, according to Equation 2.1, $\mathbf{U} = 0.625$. Also, since $\mathbf{U} \leq n(2^{\frac{1}{n}} - 1)$ the task set can be considered as schedulable. This is illustrated in Figure 2.7. Observe that no task miss its deadline. Down arrows are suppressed since they coincides with up arrows.

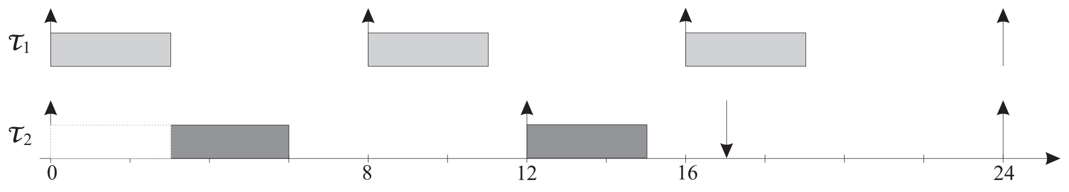


Figure 2.7. RM schedule for Example 2.1 considering $\mathbf{D} = \mathbf{T}$

For dynamic priorities scheduling policies such as EDF, the schedulability analysis can be exact [35] if task periods are equal to their deadlines ($\mathbf{D} = \mathbf{T}$). In this case the task set is said to be schedulable if and only if:

$$\mathbf{U} \leq 1 \tag{2.3}$$

2.1.3.2 Response Time Analysis

Response Time Analysis, as the name describes, is based on deriving the maximum response time R_i that a task τ_i can achieve, considering a certain scheduling policy. The basic idea of such an analysis is to calculate, for each system task, the maximum time elapsed from its release time until the end of its execution, taking into consideration the maximum interference that such a task may suffer from others, with priority greater than or equal to its own priority. To make things clear, observe Figure 2.8, a fixed priority system with two tasks.

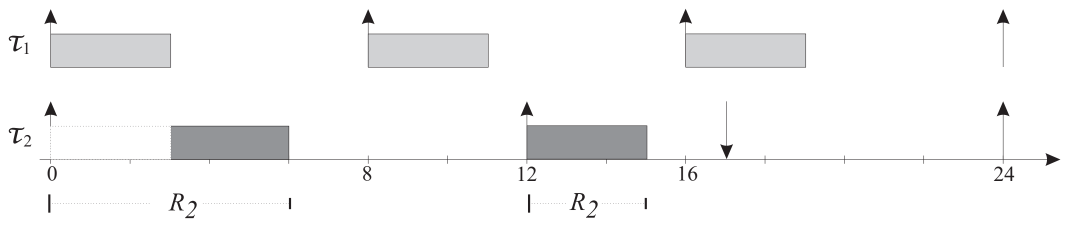


Figure 2.8. Maximum response time for τ_2 in Example 2.1 considering $\mathbf{D} = \mathbf{T}$

Observe that the first job of τ_2 is released at time 0 and finishes its execution by time 6. Thus, the response time in this case is 6. The second job of τ_2 is released at time 12 and finished its execution by time 15. Considering all jobs released during the hyperperiod h, the maximum response time for τ_2 is $R_2 = 6$. Also, notice that the first job suffers an interference from τ_1, which have an execution priority greater than that of τ_2. Since $\mathbf{D} = \mathbf{T}$, $R_2 \leq D_2$ and τ_2 is schedulable. The same reasoning is used to compute the maximum response time for τ_1, which is $R_1 = 3$. Hence, since both tasks are schedulable, the task set Γ is considered schedulable.

It is important to observe that the maximum response time for a task $\tau_i \in \Gamma$ must be calculated considering the worst-case interference for each system task. To do so, R_i is usually computed during the *critical instant*, which is defined as the time instant in which a task will have the largest response time [35]. Considering a task set with n independent tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ scheduled according to some fixed priority scheduling policy where tasks deadlines can be smaller than or equal to their periods ($\mathbf{D} \leq \mathbf{T}$) [22], the maximum response time for each task $\tau_i \in \Gamma$ is given by:

$$R_i = C_i + \sum_{j \in hp(i)} I_j$$

where $hp(i)$ is the set of tasks with priority greater than that of τ_i and I_j is the greatest interference caused by a higher priority task. The worst-case interference a task $\tau_j \in \Gamma$ can cause in τ_i can be computed by

$$I_j = \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

where $\lceil R_i/T_j \rceil$ is the number of activations of τ_j during R_i. The maximum response time for a task τ_i can be then written as [2, 37, 22]:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.4}$$

Also, the task set Γ is considered schedulable if $\forall \tau_i \in \Gamma, R_i \leq D_i$ [2]. Response time calculation is iterative, converging to a finite value if the system utilization is not greater than one. Also, such a method is not so simple when EDF scheduled systems are considered. Indeed, since this is an exact schedulability analysis, it is able of determining all schedulable task sets for any priority oriented scheduling policy.

In the following section fault tolerance aspects for real-time systems are considered. Also, some scheduling approaches to dealing with error occurrences are presented. Most of them either adapt or extend their scheduling policies and analysis techniques to comply with such events.

2.2 FAULT-TOLERANT REAL-TIME SYSTEMS

2.2.1 Fault Tolerance Overview

When it comes to computer applications, faults are inevitable and can have several causes such as specification or implementation problems, components malfunctioning, defects or fatigue, manufacturing imperfections as well as external disturbances, such as electromagnetic interference and environmental variations [56]. Before addressing the formal definition of fault it is necessary to understand the concepts of *error* and *failure*.

Considering any computational system, a *failure* occurs when there is a transition from an expected behavior (correct) to a behavior that is not expected (incorrect). That is, a *failure* represents a deviation from system specification. The system state whose processing could lead to a *failure* is called *error* state. Formally, *faults* are physical or algorithmic causes of *errors* [56, 3].

Despite the occurrence of faults, a computational application must provide confidence in the executed operations (*dependability*). This confidence is an integrated concept that involves some attributes, such as availability, reliability, safety, integrity and maintainability [3]. Faults, errors and failures pose a threat to dependability. Real-time applications, for example, may have their schedulability affected by fault occurrence. Hence, such threats should be considered and addressed appropriately in order to avoid undesirable consequences. In this case, fault tolerance mechanisms should be implemented since they aim at keeping system correctness even in the presence of faults [3]. It is important to mention that fault tolerance mechanisms do not exclude fault preventing and fault removal techniques, specially for systems that require high dependability, such as critical real-time systems [56].

Faults can be classified according to certain criteria [3], however this work will consider the persistence criterion, according to which they can be transient or permanent. Transient faults are those that occur only during a given time, then disappear. This is the case, for example, of faults caused by electromagnetic interference. When transient faults occur repeatedly they are called intermittent faults. Permanent faults are caused by a permanent defect in a computing unit, such as the lack of connectivity between two nodes of a network.

Most techniques used for providing fault tolerance are based on spatial or time redundancy. The former implements hardware replication and therefore is more used to tolerate permanent faults [56]. The later consists of repeating the computation in time and is used in the case of permanent software faults and transients faults, which has been pointed out as the most frequent ones [24] and is the focus of this work. Thus, the usual approach for dealing with transient faults in real-time systems is based on time redundancy. This means that upon the detection of an error, an extra code is executed to provide the actions necessary to keep system correctness. Such actions, defined by the system recovery scheme, can be the re-execution of the faulty task or the execution of an alternative version.

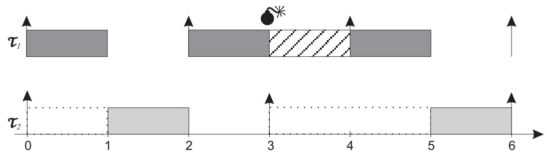
In general, two techniques for recovery, based on execution of alternative tasks, stand out: *Recovery Blocks* and *Exception Handlers.* Recovery blocks are execution blocks that detect an error by acceptance tests and backward error recovery to avoid system failures. Basically, each recovery block consists of three elements: (1) a primary module (2) an acceptance test and (3) at least one alternate module, which implements different task versions [8, 1]. Exception handlers try to undo the effects of a faulty execution, switching the execution to a specific subroutine, rather than re-executing the entire computation [8].

In the following section some scheduling approaches for fault-tolerant real-time systems are detailed. Also, their shortcomings are presented and discussed as a motivation to the following chapter.

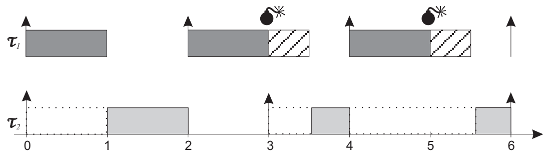
2.2.2 Temporal Redundancy Techniques for Real-Time Systems

Based on the assumption that any application potentially fails [3] fault tolerance techniques need to be used to ensure that the system will keep its correct behavior even in the presence o faults. For several real-time systems providing fault tolerance via time redundancy is effective [17, 32]. According to this approach, whenever an error is detected a recovery task is scheduled to recover the system from an incorrect state. This recovery task can be either the faulty task or any other extra code. As illustration, consider the following example.

Example 2.3 *Consider a task set $\Gamma = \{\tau_1, \tau_2\}$ composed of two independent periodic tasks scheduled by RM. Also, assume $\mathbf{C} = (1,1)$, $\mathbf{T} = (2,3)$, $\mathbf{D} = \mathbf{T}$ and that τ_1 is affected by an error at time 3. Figure 2.9 shows this schedule.*



(a) Recovery based on re-execution



(b) Recovery based on alternative task version

Figure 2.9. Fault tolerance based on time redundancy

Observe that in Figure 2.9(a) fault tolerance is achieved through the re-execution of the faulty task. In this case, the recovery execution cost $\bar{C_1} = C_1 = 1$. Also, the priority of the recovery task is the same as that of the faulty task and for this reason the recovery task preempted the execution of τ_2. In Figure 2.9(b) recovery is implemented through the execution of an alternative task version which has the same priority of the faulty task. In this case, the recovery execution cost can be smaller than the execution cost of the faulty task. In other words, $\bar{C_1} < C_1$.

Clearly, if fault tolerance techniques are considered both scheduling policy and schedulability analysis need to be adapted to take them into account. In general, schedulability analysis is built up on the assumption that the system behavior is known in worst case and this has also been true when fault tolerance aspects are considered. Nonetheless, incorporating fault tolerance aspects into schedulability is not straightforward, since errors are random events which occurrence cannot be predicted. The usual approach to taking the effects of recovery tasks into schedulability analysis is to artificially assume a worst-case pattern for errors and the derive schedulability analysis accordingly.

Several schedulability analysis for fault-tolerant real-time systems have been derived based on specific system and/or fault model. Although such approaches are important since they provide some sort of timeliness assessment, they do present some shortcomings, which will be discussed in this section. First, they are based on worst-case scenarios and do not consider the system overall behavior. Also, the derived analysis assume deterministic error patterns that may not occur and finally, most approaches are strictly linked to specific system and/or fault models which does not allow them to be compared to each of them. For example, fault tolerant approaches designed for fixed-priority scheduled systems cannot be used for dynamic-priority ones and vice versa.

For fixed-priority real-time systems, some authors fixed an upper bound on processor utilization factor for a given system Γ when it is subject to a single fault during the hyperperiod [44]. Others, considered response time analysis and fixed a minimum time between two consecutive errors [8, 17, 28]. EDF-scheduled systems have also been considered. For example, some authors assume a maximum number of errors per system task and then compute systems resilience values [32, 4, 33].

Although such approaches are important since they allow to consider error occurrence in real-time systems, they assume a worst-case error pattern which may not reflect the real system capacity to tolerate faults because they are strictly linked to the system and/or fault model. This is shown in Figure 2.9.

Observe that in Figure 2.9(a) the fault-tolerant approach is based on the re-execution and only a single fault can be tolerated in the worst case. However, if an alternative version of the faulty task is used to recover the system, two faults could be tolerated, as shown in Figure 2.9(b).

Some other authors extended schedulability analysis to consider specific fault-tolerant approaches. Sieh *et. al.* [49] assumed that different system tasks may cause different levels of injury to systems when affected by errors. Based on that the scheduler must choose to execute those jobs which bring the greatest benefit to the system as a whole.

Other approaches are characterized for pre-allocating time slices (called *slack* or *backup*) for recovery tasks, if there is an error occurrence. Han *et. al.* [25] define a solution based on Rate Monotonic and reserves slacks for alternative task versions. Ghosh *et. al.* [24] carry out schedulability analysis as suggested by Lehoczky *et. al* [31] and propose the use of extra time slices in specific time instants so that schedulability can be guaranteed. This solution can be implemented so that the most important system tasks have a time slice to be re-executed if an error occur. However, this solution may be too pessimistic, considering that worst-case scenarios hardly occur. For example, in Figure 2.9(a) one could assume that the time slice between times 5 and 6 is reserved for recovery.

The *Imprecise Computation* model was described by some authors [5, 52, 38, 41, 23] as an attempt to reducing the pessimism of worst-case analysis. In such an approach, each task τ_i can be decomposed in two parts: a mandatory part (m_i), which must be executed and is responsible for producing logically correct and accurate results, and an optional part (p_i), whose execution can be done or not. Thus, the execution cost of each task τ_i is given by $C_i = m_i + p_i$. In the case of recovery, only mandatory parts have the obligation of being executed while optional parts could be discarded. As illustration, consider the task set scheduled by RM shown in Figure 2.10 in which $m_i = p_i = 0.5$ for each task.

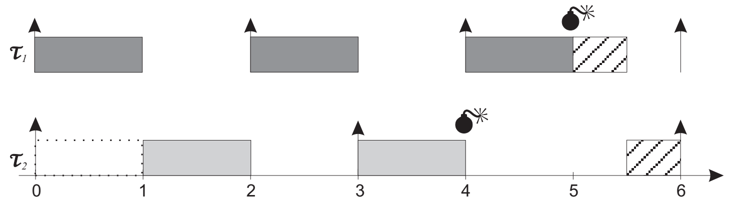


Figure 2.10. Illustration of Imprecise Computation

At time instant 5 the system begins the recovery for τ_1 according to its priority. When m_1 finishes executing, the scheduler realizes that another recovery task is ready for execution and then p_1 is discarded so that m_2 is executed.

Another approach, called *Skippable Instances*, was derived [11]. This solution is characterized by having a hybrid task set, in which the scheduler can skip executing some task instances (*skippable*) and the unused execution time can be reclaimed for recovery. Aperiodic nature of errors can also lead to an approach where some tasks are scheduled by a fixed-priority policy while others are treated by dynamic-priority ones [18, 19].

For each mentioned solution timeliness assessment is done by means of deterministic analysis techniques, which can be safely used as long as the assumptions under which such systems are designed can be predicted (eg. periodic tasks, worst-case execution time, etc.). However, such techniques may not be suitable when random events as errors are taken into consideration [45, 15]. Indeed, it can be shown that violating the assumed error pattern does not necessary imply in system failure [33].

Some probabilistic analysis methods have also been derived. However, they has not addressed fault tolerance aspects. Some authors associated a probability distribution function to tasks execution time [15, 16]. Also, based on such a distribution function, other authors determined the deadline miss probability for periodic system tasks [21, 29]. In the context of fault tolerance, response time analysis has been extended to provide probabilistic guarantees for specific fault models [7, 9], but only fixed priority scheduled systems were considered.

Despite the large number of approaches, to the best of our knowledge, there is no systematic way of comparing them. Indeed, most solutions addressed here are scheduling dependent, which means that if one is deciding to implement a given system, he/she might not be able to choose the better approach because they are not comparable. For example, solution for fixed-priority systems can not be used for dynamic-priority ones and vice-versa. Based on these shortcomings, a simulation-based approach was derived to compute a fault resilience metric for real-time systems. Such an approach is as independent of the system and/or fault model as possible. Also, it is able to represent the system overall behavior based on tasks resilience individually. Moreover, the described simulation-based analysis can be used to subsidize the system designer decisions when choosing the fault tolerant mechanisms that best suit their systems. Although other simulation-based scheduling analysis have been developed as an attempt to achieving more realistic models [55, 26], fault tolerance aspects have not been considered.

2.3 SUMMARY

This chapter presented the basic structure for real-time systems, the most used scheduling policies and schedulability analysis techniques. Also, some approaches to dealing with error occurrences in such systems were described. Although numerous, such fault-tolerant approaches for real-time system present some shortcomings which were also discussed. Based on them, a simulation-based approach was derived to compute fault resilience values for real-time systems, without taking into consideration the system and/or fault models. This approach is the focus of the following chapter.

CHAPTER 3

A SIMULATION-BASED APPROACH

This chapter derives a metric which can be used to compare different real-time systems from fault resilience viewpoint. Such a metric is reasonably independent of the system model or assumed error pattern. Also, it is capable of determining the system overall behavior based on each task. Indeed, the derived fault resilience metric must be in accordance with some desirable assumptions and requirements, which are also described here.

In order to compute such a metric, a simulation environment was built, which simulates the system during specific time windows and then compute, for each of them, the fault resilience metric. Such time windows are defined based on the concept of *simulation scenarios*, which is also presented in this chapter.

The system model and notation used to describe the simulation environment is detailed in Section 3.1. The fault resilience metric, some important requirements and assumptions are presented in Section 3.2. The developed simulation framework is presented in Section 3.3. The concept of simulation scenarios, based on which the simulation time interval is determined is detailed in Section 3.4. Finally, a summary of this chapter is presented in Section 3.5.

3.1 SYSTEM MODEL AND NOTATION

In the derived simulation-based approach it is considered uniprocessor and preemptive real-time systems composed of n periodic and independent tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$. It is assumed that $\forall \tau_i \in \Gamma, D_i \leq T_i, C_i \leq \min(T_i, D_i)$.

System schedulability can be assessed in fault-free scenarios using traditional analysis techniques (Section 2.1.3) and so only schedulable systems are considered. Fault tolerance is provided by executing an extra code upon error detection, which can be the re-execution of the faulty task or the execution of an alternative task. If errors are detected during the recovery of a task, other recovery actions can be released. Also, only transient or permanent software errors are assumed to occur. More severe types of errors, which require spatial redundancy usually implemented with a distributed/parallel architecture are not considered.

As tasks in Γ are periodic, the k-th job of task τ_i is released at time $\phi_i+(k-1)T_i, k \geq 1$, where ϕ_i is the phase of τ_i. For the sake of notation simplicity, it is assumed that $\phi_i = 0$ for all tasks in Γ, although the proposed analysis can be easily adapted to take fixed values of $\phi_i > 0$ into consideration. Aperiodic jobs are considered for error recovery only.

We restrict ourselves to systems whose scheduled jobs have fixed priorities. If J is a job of some task, $p(J)$ denotes its priority. Note that considering fixed-priority per job includes scheduling policies such as EDF, RM or DM. Hence, the proposed analysis covers a large spectrum of scheduling policies for real-time systems.

To simplify notation we define the functions $\min(\mathbf{X})$ and $\max(\mathbf{X})$, which return the minimum and maximum values of any tuple $\mathbf{X}$. For example, $\min(\mathbf{T}) = \min_{i=1}^{n}(T_i)$.

3.2 ON THE FAULT RESILIENCE METRIC

A suitable fault resilience metric must be able to measure the system resilience for different systems and fault models so that different real-time systems can be compared from resilience view point. In order to give some intuition on the need for a fault resilience metric, consider the following example.

Example 3.1 *Let Γ be a task set composed of two periodic tasks $\Gamma = \{\tau_1, \tau_2\}$. Assume that $\mathbf{T} = (2, 5)$, $\mathbf{C} = (1, 1)$, $\mathbf{D} = \mathbf{T}$ and $\bar{\mathbf{C}} = (1, 1)$. Also, consider that Rate Monotonic is used to schedule the tasks and so τ_1 is the highest priority task. Figure 3.1 shows the schedule for this task set.*

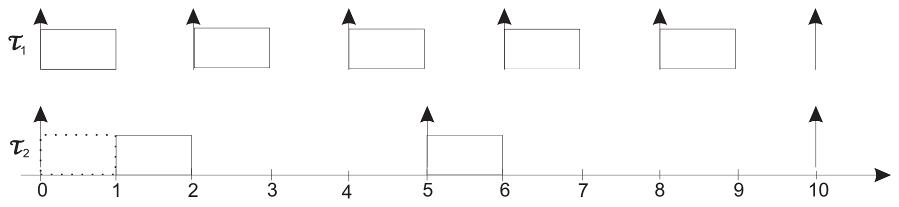
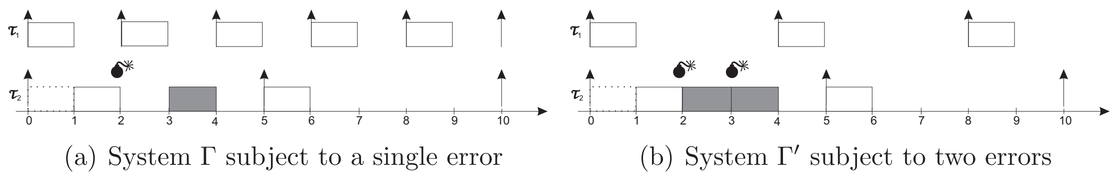


Figure 3.1. Illustration of a RM schedule for the task set described in Example 3.1

The hyperperiod for this task set is given by $h = \text{lcm}(T_1, T_2) = 10$. Notice that τ_1 has five *jobs* within h, which are released at times $0, 2, 4, 6$ and 8, while τ_2 has two jobs released at times 0 and 5. Because of its priority, the first job of τ_2 begins to execute at time 1. The dotted line indicates that τ_2 awaits until higher priority jobs finishes executing.

Assume that recovery is based on the re-execution of the faulty jobs. Figure 3.2 shows two systems that are subject to different number of errors and still meet their deadlines. Notice that right after the error takes place (at time 2 in Figure 3.2(a) and times 2 and 3 in Figure 3.2(b)), the recovery job (shown in gray) executes to keep system correctness. Figure 3.2(a) shows the schedule for the task set shown in Example 3.1 and considers a single error occurrence. Figure 3.2(b) shows the schedule for the same task set presented in Example 3.1 assuming that T_1 has changed to 4. Observe that within the first 5 time units, τ_2 can tolerate just one error in Figure 3.2(a), while this same task in Figure 3.2(b) can tolerate two.



(a) System Γ subject to a single error (b) System Γ' subject to two errors

Figure 3.2. Illustration of two different systems subject to errors within h

The schedules shown in Figure 3.2 give the intuition that the fault resilience metric must reflect the number of errors the system can tolerate. Thus, assuming that one wishes to analyze the behavior of a given system $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$ when it is subject to faults, the analysis method must use some fault resilience metrics. We understand that such metrics must take the following assumption into consideration.

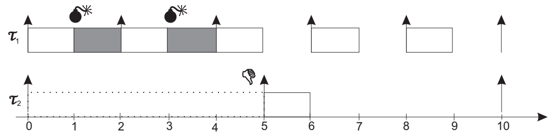
Assumption 3.2.1 *The fault resilience of a system is proportional to the number of errors it tolerates.*

Indeed, fault resilience metrics must reflect the system ability to survive, or keep its correct behavior, after error occurrences. Most authors are aware of that [17, 44, 32, 41], despite assuming specific error patterns. Also, from designers' view point it is important to determine how many errors can occur before a system failure.

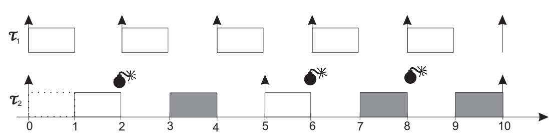
In several situations, the number of error occurrences accounted for when analyzing a system must be a function of time. The intuition is that the expected number of errors increases with time assuming that they are not co-related. Although some authors do not consider such assumption [44, 25], some fault models in real-time systems that are in line with this observation can be mentioned, as for example Poisson distribution [7] and minimum time between errors [8, 9]. Based on that, we assume the following:

Assumption 3.2.2 *The expected number of error occurrences increases with time.*

Since the system we are considering is composed of n tasks and each of them may have a different level of criticality, the fault resilience metric must be determined for each individual task so that system designers can deal with the peculiarities of each of them. Figure 3.3 shows the maximum number of errors that two different tasks from the same system can cope with.



(a) Maximum number of errors for $\tau_1 \in \Gamma$



(b) Maximum number of errors for $\tau_2 \in \Gamma$

Figure 3.3. Illustration of two different tasks of Γ subject to errors within h

Considering Example 3.1, τ_1 can timely recover from at most 1 errors within 2 time units, while τ_2 can deal with one or two errors within 5 time units. This is shown in Figures 3.3(a) and 3.3(b), respectively. Note that in the figure errors take place at the end of the job execution, the worst-case scenario. Thus, as tasks may have different criticality levels, it is interesting to consider the following requirement:

Requirement 3.2.1 *Fault resilience must be given for individual tasks of the analyzed system.*

Further, the resilience of a given task $\tau_i \in \Gamma$ depends on how its jobs behave when errors take place. For example, two jobs of τ_i might be capable of tolerating different number of error occurrences during their executions, due to the different interferences they suffer. Observe in Figure 3.3(b) that the first instance of τ_2 can tolerate only a single error. Since τ_1 interferes in its execution, there is not enough time to recover the system from two errors. On the other hand, the second job of τ_2 can tolerate two errors, since the interference it suffers from τ_1 is smaller. Indeed, scheduling decisions or the set of interfering jobs may not be the same for all jobs of τ_i. In order to capture the behavior of τ_i as a whole, therefore, different jobs of the analyzed task must be taken into consideration. This motivates the following requirement:

Requirement 3.2.2 *The fault resilience of a task must account for the overall behavior of its jobs.*

Based on the assumptions and requirements stated above, we give the following definition of fault resilience metric:

Definition 3.1 *The fault resilience of a job is measured as the minimum number of errors that makes it miss its deadline divided by its relative deadline. The fault resilience distribution of a task is given by the fault resilience of its jobs.*

According to the above definition, error occurrences are considered for each task in a per-job basis, which is in line with Requirements 3.2.1 and 3.2.2. Also, the time windows in which jobs execute (relative deadlines) are taken into consideration. Indeed, the longer the execution of a job the more likely errors occur, which complies with Assumption 3.2.2. Finally, observe that Assumption 3.2.1 is also considered.

Obviously, different metrics can be given. Some assumptions/requirements may not be suitable for all systems while new assumptions/requirements may be needed for others. For example, if Requirement 3.2.2 is not needed, usual analysis based on worst-case scenarios may suffice. Further, if Assumption 3.2.2 is removed, counting the minimum number of errors per task in worst-case is enough [33]. In any case, we stress here that we use a metric which is in line with Definition 3.1 as a means of fault resilience assessment.

Motivated by the above requirement and assumptions, we derive a fault resilience analysis based on simulation, as will be explained in the following section.

3.3 SIMULATION ENVIRONMENT

The scheme of the proposed simulation-based analysis method is illustrated in Figure 3.4. It is based on modeling a fault tolerant real-time system so that system execution is represented by two main components: the *scheduler* and the *error generator*. While the former follows a given scheduling policy and tries to keep the system schedulable, the goal of the latter is to generate faults so that task deadlines are missed. Thus, the error generator acts as an adversary to the scheduler. No particular error pattern is assumed. A role of the error generator is to derive the worst-case error pattern for each simulation. These two components are named *simulation engine*.

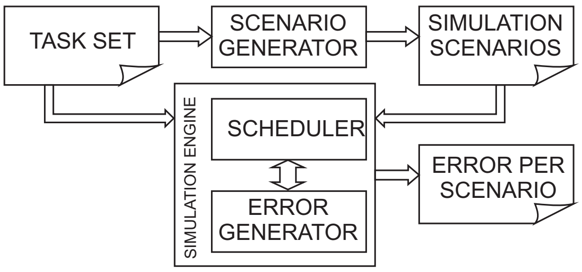


Figure 3.4. Simulation Environment

The general idea of the simulation-based analysis is to generate possible tuples of task release times, which we call *simulation scenarios* and for each tuple compute the minimum number of errors that makes a specific job miss its deadline. Then, considering this number of errors per scenario and the simulation time interval, the fault resilience metric is computed.

As mentioned before, as long as simulation scenarios have been generated the simulation engine computes the fault resilience metric for each scenario. Considering that $f_i^{\mathbf{S}}$ is the minimum number of errors that makes a task τ_i unschedulable in a given simulation scenario $\mathbf{S}$, the effort value is defined as follows:

Definition 3.2 *Let $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be a task set and consider $\mathbf{S}$ a simulation scenario of Γ. The effort $\mathbf{E}_i$ made by the error generator to make a task $\tau_i \in \Gamma$ unschedulable in $\mathbf{S}$ is:*

$$\mathbf{E}_i = \frac{f_i^{\mathbf{S}}}{D_i} \tag{3.1}$$

Notice that the effort definition in accordance with Definition 3.1. Also, considering the effort for several simulation scenarios allows one to infer the overall behavior of jobs from fault resilience view point, as stated in Requirement 3.2.2. Intuitively, the higher the effort $\mathbf{E}_i$ made by the error generator, the higher the resilience of τ_i regarding scenario $\mathbf{S}$. Note that the found values of $\mathbf{E_i}$ can be statistically analyzed. Also, one may be interested in other parameters, such as the effort mean value $\bar{\mathbf{E}}_i$ or the minimum effort $\mathbf{E}_i^{\min}$ necessary to make a given system task unschedulable. Analyzing the simulation data may make it possible to derive the parameter of interest.

It is important to observe that for complex systems determining $\mathbf{E}_i$ or any function of it may be too time consuming, since the total number of simulation scenarios is a function of the hyperperiod of the task set. For this reason, this simulation environment considers studying $\mathbf{E}_i$ through sampling. The simulation time interval in which $\mathbf{E}_i$ is computed is derived based on the concept of simulation scenarios, which are the focus of the following section.

3.4 SIMULATION SCENARIOS

Unlike the existing approaches, the simulation environment illustrated in Figure 3.4 does not need to simulate the whole execution of the system, which might be too timing consuming in the general case. The idea is to simulate specific time windows and then derive system fault resilience by statistically analyzing the simulation data. These time windows are obtained through the concept of simulation scenarios, defined by the *scenario generator*. As a motivation to the content of this section, consider the following illustrative example:

Example 3.2 *Let* $\Gamma = \{\tau_1, \tau_2, \tau_3\}$ *be a set of periodic tasks scheduled according to EDF. The task parameters are* $\mathbf{T} = (10, 15, 20)$, $\mathbf{C} = (4, 4, 4)$ *and* $\mathbf{D} = \mathbf{T}$. *The hyperperiod of this task set is defined as* $h = \mathrm{lcm}(10, 15, 20) = 60$. *Figure 3.5 shows the EDF schedule for* Γ *within* $[0, 60)$ *assuming that all tasks are released at time zero.*

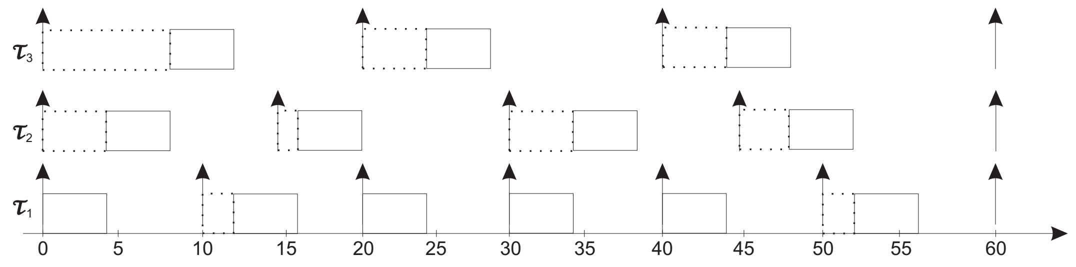


Figure 3.5. Illustration of an EDF schedule for the task set described in Example 3.2

Considering that tasks always take their worst-case execution times to complete and that they are all released at some time t, a simple and usual approach to simulating a schedule has to take into account the time interval window $[t, t + h)$, where h is the hyperperiod of the task set. By doing so, one can represent all possible scenarios, relative to the assumed system start time, by simulating the schedule as in Example 3.2. As mentioned before, this approach may not be practical since the hyperperiod may be too large. The approach used here divides the hyperperiod into specific time windows, in order to represent only parts of the schedule.

For example, considering that one wishes to analyze the behavior of the second instance of τ_3, a possible simulation window could be $[20; 40)$. If a large enough number of such windows are selected to check the behavior of the jobs of τ_3, after observing the schedule of them, one can infer the average behavior of τ_3. It can be observed that, determining simulation windows depends on the release time of jobs.

Obviously, selecting arbitrary simulation windows brings about some interesting and important questions. For example, why $[20, 40)$ is a representative time window? Why not $[30, 45)$ or $[0, 20)$? How can one simulate a specific time window without considering what happens before the window? And finally, how such a kind of window can be selected? Notice that if the number of different scenarios is not large enough, it is likely that a schedule can be simulated as in Example 3.2. Otherwise, suitable strategies to determine these windows must be derived.

The following sections answer some of the questions stated above. Nonetheless, some of the issues raised in this chapter will be left to be addressed in the next chapters. The focus of the remainder of this chapter is on precisely defining the concept of *simulation scenarios*.

3.4.1 The Concept of Simulation Scenarios

Considering a periodic task set, a representation of a possible execution of the system can be given by tasks release times. For example, in the schedule represented in Figure 3.5, tuples such as $(0, 0, 0)$ or $(10, 15, 0)$ are configurations of release times for τ_1, τ_2 and τ_3, respectively. Based on this kind of tuples the starting and finishing times of any simulation can be defined using some other criteria, as will be seen in Chapter 4. This concept is given more precisely as follows:

Definition 3.3 *Tuple* $\mathbf{S} = (S_1, \ldots, S_n)$ *is a simulation scenario of a periodic task set*

$\Gamma = \{\tau_1, \ldots, \tau_n\}$ *if the following predicate holds:*

$$\text{scenario}(\Gamma, \mathbf{S}) \overset{\text{def}}{=} \exists w \in \quad, \forall S_i : (S_i + w) \mod T_i = 0 \quad \wedge \quad \max(\mathbf{S}) - S_i < T_i \tag{3.2}$$

Both conditions defined by the above predicate mean that: (a) $\mathbf{S}$ is a tuple of tasks release times; and (b) only the closest jobs, released before the last released job, are considered. Taking Example 3.2 as illustration, it can be seen that according to Definition 3.3 tuples $(0, 0, 0), (20, 15, 20)$ and $(40, 30, 40)$ are simulation scenarios. However, tuple $\mathbf{S} = (40, 15, 40)$, say, is not. In this example, although S_i is a possible release time of τ_i $(i = 1, 2, 3)$, the release time of τ_2 should be 30 instead of 15 to make $\mathbf{S}$ a simulation scenario for this task set example.

Consider tuples $\mathbf{S} = (20, 15, 20)$ and $\mathbf{S}' = (30, 25, 30)$, say, both simulation scenarios for Example 3.2. Note that $S_i = S'_i + 10$ for all tuple elements S_i. This means that the same simulation effects would be observed when $\mathbf{S}$ or $\mathbf{S}'$ were simulated. In this case, it is said that $\mathbf{S}$ is equivalent to $\mathbf{S}'$:

Definition 3.4 *Two scenarios* $\mathbf{S}$ *and* $\mathbf{S}'$ *of a periodic task set* $\Gamma = \{\tau_1, \ldots, \tau_n\}$ *are equivalent to each other if and only if* $S_i - S'_i$ *is constant,* $i = 1, \ldots, n$. *More formally,*

$$\mathbf{S} \equiv \mathbf{S}' \Leftrightarrow \exists w \in \quad, \forall i \in \{1, \ldots, n\} : S_i = S'_i + w \tag{3.3}$$

Indeed, a set of simulation scenarios of a periodic task set Γ must represent possible release time distances between jobs of distinct tasks of Γ. Simulating equivalent simulation scenarios must be avoided. Therefore, the set of all distinct (non-equivalent) simulation scenarios need to be defined:

Definition 3.5 *The set of all possible simulation scenarios of* Γ *is denoted* Ω *and is defined such that: (a) all elements of* Ω *are simulation scenarios of* Γ*; (b) there is no distinct scenarios in* Ω *equivalent to each other; (c) any element not in* Ω *is not a scenario* *or is equivalent to some scenario in* Ω. *This is formally expressed by:*

$$\begin{aligned} \forall \mathbf{S}, \mathbf{S}' \in \Omega, \forall \mathbf{S}'' \notin \Omega, \exists \mathbf{S}''' \in \Omega : \text{scenario}(\Gamma, \mathbf{S}) \wedge \text{scenario}(\Gamma, \mathbf{S}') \wedge \\ (\mathbf{S} \equiv \mathbf{S}' \Leftrightarrow \mathbf{S} = \mathbf{S}') \wedge \\ (\neg \textbf{scenario}(\Gamma, \mathbf{S}'') \vee \mathbf{S}''' \equiv \mathbf{S}'') \end{aligned} \tag{3.4}$$

Considering Example 3.2, the set of all possible scenarios can be $\Omega = \{(0,0,0), (10,0,0), (20,15,20), (30,30,20), (40,30,40), (40,45,40), (50,45,40)\}$. Each tuple $\mathbf{S} \in \Omega$ is obtained based on a specific task $\tau_i \in \Gamma$. Suppose one wishes to analyze the jobs of τ_i only. It is worth defining a set which contains simulations scenarios relative to a specific task τ_i. This is defined as follows:

Definition 3.6 *The set of all distinct simulation scenarios of a task* $\tau_i \in \Gamma$ *is denoted* Ω_i *and can be obtained by:*

$$\Omega_i = \{\mathbf{S} \in \Omega | \exists w \in \quad : \max(\mathbf{S}) + w \mod T_i = 0\}$$

For Example 3.2, $\Omega_1 = \{(0,0,0), (10,0,0), (20,15,20), (30,30,20), (40,30,40), (50,45,40)\}$, $\Omega_2 = \{(0,0,0), (10,15,0), (30,30,20), (40,45,40)\}$ and $\Omega_3 = \{(0,0,0), (20,15,20), (40,30,40)\}$. It is clear that $\bigcup_{i=1}^{n} \Omega_i = \Omega$.

3.4.2 Useful Operations on Simulation Scenarios

Some operations on simulation scenarios will be used when deriving scenario generation procedures. First, it is convenient to define the time-shift operation. Actually, it shifts the time-axis so that an equivalent scenario is produced.

Definition 3.7 *The time-shift operation executed on a scenario of a periodic task set*

$\Gamma = \{\tau_1, \ldots, \tau_n\}$ *is defined by*

$$\text{tshift}(\mathbf{S}, w) \overset{\text{def}}{=} (S_1 + w, \ldots, S_n + w), \quad w \in \tag{3.5}$$

It is clear that $\mathbf{S} \equiv \text{tshift}(\mathbf{S}, w)$, for any $w \in$.

Consider Example 3.2 and one of its simulation scenarios represented in Figure 3.5, say $\mathbf{S} = (20, 15, 20)$. This scenario takes place after time advances by at least 20 time units from the origin. Advancing time by 10 time units further, scenario $(30, 30, 20)$ is reached. This reasoning suggests the following useful operation on $\mathbf{S}$ that gives the scenario that is found from $\mathbf{S}$ if time advances by a given value.

Definition 3.8 *Let $\Gamma = \{\tau_1, \ldots, \tau_n\}$ be a set of periodic task set and $\mathbf{S}$ a simulation scenario of Γ. The time-add operation on $\mathbf{S}$ is defined as*

$$\text{tadd}(\Gamma, \mathbf{S}, w) \overset{\text{def}}{=} \left(S_1 + \left\lfloor \frac{\max(\mathbf{S}) + w - S_1}{T_1} \right\rfloor T_1, \ldots, S_n + \left\lfloor \frac{\max(\mathbf{S}) + w - S_n}{T_n} \right\rfloor T_n \right) \tag{3.6}$$

The result of the time-add operation is also a scenario, as stated by the following lemma.

Lemma 3.1 *Let $\Gamma = \{\tau_1, \ldots, \tau_n\}$ be a set of periodic tasks and $\mathbf{S}$ a simulation scenario of Γ. For any time interval $w \geq 0$, $\mathbf{S}' = \text{tadd}(\Gamma, \mathbf{S}, w)$ is also simulation scenario of Γ.*

Proof Consider that $\mathbf{S}$ is a simulation scenario of a periodic task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$. It is clear that $S'_i \mod T_i = 0$ since $S_i \mod T_i = 0$. Also, the following relation holds:

$$\max(\mathbf{S}) + w - S_i - \left\lfloor \frac{\max(\mathbf{S}) + w - S_i}{T_i} \right\rfloor T_i < T_i,$$

which implies that $\max(\mathbf{S}) + w - S'_i < T_i$. As $\max(\mathbf{S}) + w < \max(\mathbf{S}')$, relation $\max(\mathbf{S}') - S'_i < T_i$ must hold for $i = 1, \ldots, n$. Therefore, $\mathbf{S}'$ is a simulation scenario of Γ according to Definition 3.3. ▮

Similarly, the operation $\mathrm{tsub}(\Gamma, \mathbf{S}, w)$, which gives a scenario when time goes back by w, can be defined:

Definition 3.9 *Let $\Gamma = \{\tau_1, \ldots, \tau_n\}$ be a set of periodic tasks and $\mathbf{S}$ a simulation scenario of Γ. The time-sub operation on $\mathbf{S}$ is defined as*

$$\mathrm{tsub}(\Gamma, \mathbf{S}, w) \overset{def}{=} \left(S_1 + \left\lfloor \frac{\max(\mathbf{S}) - w - S_1}{T_1} \right\rfloor T_1, \ldots, S_n + \left\lfloor \frac{\max(\mathbf{S}) - w - S_n}{T_n} \right\rfloor T_n \right) \tag{3.7}$$

It is important to notice that $\mathrm{tsub}(\Gamma, \mathbf{S}, w) \equiv \mathrm{tadd}(\Gamma, \mathbf{S}, -w)$. Thus, Lemma 3.1 can be used to show that result of the time-sub operation is also a simulation scenario.

It is interesting to observe that $\mathrm{tadd}(\Gamma, \mathbf{S}, w)$ is a step-functions which changes its value whenever $\max(\mathbf{S}) + w - S_i$ are multiple of T_i. Taking Example 3.2 as illustration, it can be seen that using any value in $[0, 5)$ for w leads to the same simulation scenario since from any $\mathbf{S} \in \Omega$, $\mathrm{tadd}(\Gamma, \mathbf{S}, w) = \mathrm{tadd}(\Gamma, \mathbf{S}, 0)$. Further, it is not difficult to check that $\mathrm{tadd}(\Gamma, \mathbf{S}, w) \equiv \mathrm{tadd}(\Gamma, \mathbf{S}, w + 60)$ since $\mathrm{lcm}(10, 15, 20) = 60$. These observations imply that there is a finite set of values for w that can be used to generate simulation scenarios for a periodic task set. Following this arguments, the following theorem gives a basis for deriving scenario generation procedures for a given periodic task set.

Theorem 3.1 *Consider a periodic task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$ and its hyperperiod defined as $h = \mathrm{lcm}(T_1, \ldots, T_m)$. If $\mathbf{S}$ is a simulation scenario of Γ, then the set of all its (non-equivalent) simulation scenarios Ω can be obtained by*

$$\Omega = \bigcup_{w \in \Psi} \mathrm{tadd}(\Gamma, \mathbf{S}, w), \tag{3.8}$$

where

$$\Psi = \{v \in [0, h) | \exists \tau_i \in \Gamma : v \mod T_i = 0\}$$

Proof From Lemma 3.1, it is known that $\mathrm{tadd}(\Gamma, \mathbf{S}, w)$ is a simulation scenario of Γ. It is needed to show that such values of w give Ω. Consider any value of $w \notin \Psi$. The proof will be by showing that $\mathrm{tadd}(\Gamma, \mathbf{S}, w) \equiv \mathrm{tadd}(\Gamma, \mathbf{S}, v)$ for some $v \in \Psi$. First, consider that $v \mod T_i \neq 0$ for any $\tau_i \in \Gamma$. In this case, write $v = v_i + x_i$ such that $v_i \mod T_i = 0$ and $0 < x_i < T_i$ for all $\tau_i \in \Gamma$. Let $v' = \max_{\tau_i \in \Gamma}(v_i)$. Since $\mathbf{S}$ is a simulation scenario of Γ, it follows that $\lfloor(\max(\mathbf{S}) + v' - S_i)/T_i\rfloor = \lfloor(\max(\mathbf{S}) + v - S_i)/T_i\rfloor$ for all $\tau_i \in \Gamma$. This means that $\mathrm{tadd}(\Gamma, \mathbf{S}, v') = \mathrm{tadd}(\Gamma, \mathbf{S}, v)$. If $v \in [0, h)$, let $w = v' \in \Psi$, which is enough to show the theorem. Otherwise, let $w' \in$ be a value such that $w = v' + w'$ and $w \in [0, h)$. It is clear that such w' exists. Thus, by Definition 3.4, $\mathrm{tadd}(\Gamma, \mathbf{S}, w) \equiv \mathrm{tshift}(\mathrm{tadd}(\Gamma, \mathbf{S}, v'), w')$, completing the proof. ▮

Corollary 3.1 *Let* $\mathbf{S} = (0, \ldots, 0)$ *be a simulation scenario. The set of all non-equivalent distinct simulation scenarios for a given task* τ_i *can be obtained by*

$$\Omega_i = \bigcup \mathrm{tadd}(\Gamma, \mathbf{S}, kT_i),\ k = 0, 1, \ldots, \frac{h - T_i}{T_i}$$

Proof Assume $\mathbf{S}' = \mathrm{tadd}(\Gamma, \mathbf{S}, kT_i)$ for some $k \in$ and $0 \leq k \leq (h - T_i)/T_i$. From Lemma 3.1, it is known that $\mathbf{S}'$ is a simulation scenario of Γ. As $\mathbf{S} = (0, \ldots, 0)$, $\mathbf{S}' = \left(\left\lfloor \frac{kT_i}{T_1} \right\rfloor T_1, \ldots, \left\lfloor \frac{kT_i}{T_n} \right\rfloor T_n\right)$, by Equation (3.6). From Theorem 3.1, with these values of k, a subset of Ω is generated. Hence, it is necessary to show that such a subset is Ω_i, which is equivalent to showing that $\mathbf{S}' \in \Omega_i$. Assume by contradiction that $\mathbf{S}' \notin \Omega_i$. By Definition 3.6, this means that $\max(\mathbf{S}') = \left\lfloor \frac{kT_i}{T_l} \right\rfloor T_l$ for some $\tau_l \neq \tau_i$, or equivalently,

$$\left\lfloor \frac{kT_i}{T_i} \right\rfloor T_i < \left\lfloor \frac{kT_i}{T_l} \right\rfloor T_l \Rightarrow \frac{kT_i}{T_l} < \left\lfloor \frac{kT_i}{T_l} \right\rfloor,$$

and so the corollary follows since $\frac{kT_i}{T_l} \geq 0$. ▮

As an illustration of Theorem 3.1, consider $\mathbf{S} = (0, 0, 0)$ regarding Example 3.2. It can be seen that Ω can be given by $\mathrm{tadd}(\Gamma, \mathbf{S}, w)$ for any $w \in \Psi = \{0, 10, 15, 20, 30, 40, 45, 50\}$. Considering task τ_1, Ω_1 can be given by $\mathrm{tadd}(\Gamma, \mathbf{S}, kT_1)$ for any $k \in \lambda = \{0, 1, 2, 3, 4, 5\}$. Although simple, the observations stated by both the theorem and the corollary are important for deriving algorithms to generate simulation scenarios. This is the main idea of most algorithms given in the next chapter. Also, it is important to observe that both Theorem 3.1 and Corollary 3.1 can be extended to consider the generation on Ω and Ω_i regarding tsub() operation.

3.5 SUMMARY

This chapter presented a fault resilience metric which is reasonably independent of the system and fault model. Such a metric can be used to compare different systems from the resilience point of view. In order to compute this metric, a simulation environment was defined so that the system can be simulated during specific time windows, which are defined based on simulation scenarios, a concept formalized in this chapter.

The definitions presented in this chapter are a basis to deriving procedure to generate simulation scenarios and those to generate errors during simulation. These issues will be addressed in the next chapters.

CHAPTER 4

SIMULATION SCENARIO GENERATION PROCEDURES

This chapter focus on describing the generation procedures for simulation scenarios. Some of the derived algorithms, presented in Section 4.1, generate all possible simulation scenarios for a given task. Others, described in Section sec:randomscenario, randomly generate a subset of them. Also, the time complexity for each procedure is presented for comparison means.

4.1 SEQUENTIAL SCENARIO GENERATION PROCEDURES

Sequential generation procedures are characterized by generating the set of all possible simulation scenarios Ω_i for a given task $\tau_i \in \Gamma$. More specifically, the procedures described here must comply with the following properties:

Property 4.1 (Full-coverage) *All simulation scenarios Ω_i are generated.*

Property 4.2 (Validity) *Only simulation scenarios are generated.*

Property 4.3 (Termination) *The generation procedure takes a finite number of steps.*

Two procedures that satisfy these three properties are described. The first one is based on the knowledge of the task set hyperperiod, and is described in Section 4.1.1. The second procedure, described in Section 4.1.2, is capable of generating Ω_i without the need to determine the hyperperiod.

4.1.1 A Simple Sequential Generation Procedure

Algorithm 1 sequentially generates the set Ω_i starting from $\mathbf{S} = (0, \ldots, 0)$. The generation procedure receives T_i as input and advances by steps of T_i. A new scenario is generated in every step and $k = 0, 1 \ldots, (h - T_i)/T_i$. Note that $\mathbf{S} = (0, \ldots, 0)$ is fixed during the execution of the algorithm.

Algorithm 1: A simple procedure to generate Ω_i for a periodic task set.

```
1 $k \leftarrow 0$;
2 $h \leftarrow \mathrm{lcm}(T_1, \ldots, T_n) - min_{j=1}^{n}(T_j)$;
3 $\Omega \leftarrow \emptyset$;
4 $\mathbf{S} \leftarrow (0, \ldots, 0)$;
5 while $k < (h - T_i)/T_i$ do
6     $\Omega \leftarrow \Omega \cup \mathrm{tadd}(\Gamma, \mathbf{S}, kT_i)$;
7     $k \leftarrow k + 1$;
```

Table 4.1 illustrates the behavior of the algorithm when it is applied for Example 3.2, considering task τ_1. Recall that $\mathbf{T} = (10, 15, 20)$. As can be seen, 6 values of k are used by the algorithm.

Table 4.1. Results of Algorithm 1 for illustrative example.

k	$\mathrm{tadd}(\Gamma, \mathbf{S}, kT_i)$	k	$\mathrm{tadd}(\Gamma, \mathbf{S}, kT_i)$	k	$\mathrm{tadd}(\Gamma, \mathbf{S}, kT_i)$
0	$(0, 0, 0)$	2	$(20, 15, 20)$	4	$(40, 30, 40)$
1	$(10, 0, 0)$	3	$(30, 30, 20)$	5	$(50, 45, 40)$

It is not difficult to see the correctness of Algorithm 1:

Theorem 4.1 *Algorithm 1 generates only simulation scenarios for any set of periodic tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ and all possible simulation scenarios Ω_i, regarding a given task $\tau_i \in \Gamma$, are generated in a finite number of steps.*

Proof The validity porperty follows directly from line 6 and Lemma 3.1. Also, it can be seen by the algorithm that there are exactly h/T_i steps and so the Termination property holds. By the algorithm, only simulation scenarios within the interval $[0, h)$ are generated.

Further, by Corolary 3.1, kT_i, $k = 0, \ldots, (h - T_i)/T_i\}$ ensures the Full Coverage property, completing the proof. ▮

Note that Algorithm 1 uses the value of the task hyperperiod, which can be calculated efficiently by successively performing the classical Euclid's algorithm [30]. Even though this is true, its value may grow rapidly and generate large numbers. Hence, depending on the relations between task periods, the computation of the hyperperiod can be time/memory consuming due to arithmetics on large numbers that must be carried out [14] during the generation procedure.

Traditional algorithms used to do such arithmetics consider that the multiplication and division operation take $O(\beta^2)$, where β is the number of bits used to save each number. Better algorithms have already been developed, however, in practice assuming $O(\beta^2)$ generally presents a better result [14]. For example, considering a task set Γ with n tasks, the running time for the hyperperiod computation is $O((n-1)(\log_2 h)^2)$ [30]. Furthermore, the algorithm may have to deal with large numbers arithmetics in **time-add** operation. Hence, the running time for Algorithm 1 is $O(\frac{nh}{T_i}(\log_2 h)^2)$.

It may be preferable to use algorithms that do not depend on the knowledge of task set hyperperiod. An algorithm that avoids working with large numbers is shown in the following section.

4.1.2 A Better Sequential Generation Procedure

The main idea of Algorithm 2 is to carry out time-shift operations during the generation procedure so that one of the release times in the simulation scenarios are always kept at the origin. Recall that simulation scenarios are equivalent with respect to time-shift operations (Definition 3.4). The algorithm starts from scenario $(0, \ldots, 0)$ and advances in steps of T_i. Tuple $\mathbf{S}$ is updated in line 6 so that each generated $\mathbf{S}$ have S_1 at the origin. Any S_i could be chosen instead of S_1.

Algorithm 2: A better procedure to generate Ω_i for a periodic task set.

```
1 $\mathbf{S} \leftarrow (0, \ldots, 0)$;
2 $\Omega \leftarrow \emptyset$;
3 repeat
4     $\Omega \leftarrow \Omega \cup \mathbf{S}$;
5     $\mathbf{S} \leftarrow \text{tadd}(\Gamma, \mathbf{S}, T_i)$;
6     $\mathbf{S} \leftarrow \text{tshift}(\mathbf{S}, -S_1)$;
7 until $S_1 = S_2 = \ldots = S_n$ ;
```

Taking Example 3.2 for illustration and considering task τ_1, the steps of Algorithm 2 are represented in Table 4.2. The second and third columns of the table show the simulation scenarios generated before and after the time-shift operations (line 6), respectively. It is interesting to observe that the release time values are bounded within a few values unlike Algorithm 1. For this example, this strategy keeps the values of S_i between -10 and 5 instead of within $[0, h)$. As a consequence, time and space related to the computation of large numbers associated to the task set hyperperiod are saved. The last column of Table 4.2 indicates the equivalences between the scenarios generated by Algorithm 2 and the one shown in Table 4.1. The algorithm halts in the seventh step, when scenario $\mathbf{S} = (0, 0, 0)$ is reached for the second time.

Table 4.2. Results of Algorithm 2 for Example 3.2

step	$\text{tadd}(\Gamma, \mathbf{S}, v)$	$\text{tshift}(\mathbf{S}, -v)$	equiv.
1	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$
2	$(10,0,0)$	$(0,-10,-10)$	$(10,0,0)$
3	$(10,5,10)$	$(0,-5,0)$	$(20,15,20)$
4	$(10,10,0)$	$(0,0,-10)$	$(30,30,20)$
5	$(10,0,10)$	$(0,-10,0)$	$(40,30,40)$
6	$(10,5,0)$	$(0,-5,-10)$	$(50,45,40)$
7	$(10,10,10)$	$(0,0,0)$	halt

The correctness of Algorithm 2 is now derived, where the Validity, Full Coverage, Termination properties are shown.

Theorem 4.2 *Algorithm 2 generates only simulation scenarios for any set of periodic tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ and all possible simulation scenarios Ω_i, regarding a given task*

$\tau_i \in \Gamma$, *are generated in a finite number of steps.*

Proof Assume that $\mathbf{S}$ and $\mathbf{S}'$ are simulation scenarios generated (in line 6) in two consecutive steps of the algorithm. The properties will now be shown:

Validity. By a simple induction on the scenarios generated in each step of the algorithm. As the base case, let $\mathbf{S} = (0, \dots, 0)$. By definition, it is clear that $\mathbf{S}$ is a simulation scenario. It follows from lines 5,6 and from Lemma 3.1 that $\mathbf{S}'$ is also a simulation scenario. Assuming now that $\mathbf{S}$ is a simulation scenario generated by the algorithm in any of its steps, it can be shown that $\mathbf{S}'$ is a simulation scenario by the same arguments used for the base case.

Termination. It is necessary to show that the halting state $S'_1 = S'_2 = \dots = S'_n$ is eventually reached. Since it is known by Definition 3.7 that time-shift operations produce equivalent scenarios, removing line 6 does not alter the algorithm correctness. Carrying out this modification, the same halting state is reached when

$$\left\lfloor \frac{kT_i}{T_1} \right\rfloor = \left\lfloor \frac{kT_i}{T_2} \right\rfloor = \dots = \left\lfloor \frac{kT_i}{T_n} \right\rfloor$$

where $k > 0$ is the number of times T_i is added. This identity can take place only if $kT_i = h = \operatorname{lcm}(T_1, \dots, T_n)$, i.e. $k = 0, 1, \dots, h/T_i$ and so the algorithm terminates in h/T_i steps.

Full Coverage. Assume by contradiction that there is a scenario $\mathbf{S}'' \in \Omega_i$ not generated by the algorithm. From Equation (3.6), it is known that $\mathbf{S} \neq \mathbf{S}'$ and by line 5, the procedure advances in steps of T_i. In other words, no simulation scenario for τ_i is skipped during the generation procedure. Thus, the algorithm must have halted before generating $\mathbf{S}''$. As mentioned before, k assumes integer values between 0 and $\frac{h-T_i}{T_i}$, which by Corollary 3.1 ensures full coverage.

∎

It is important to observe that Algorithm 2 does not use large numbers and hence it saves the cost of calculating them. Also, as mentioned before, it deals with numbers bounded within a few values. For these reasons, the running time of Algorithm 2 is $O(nh/T_i)$.

4.2 RANDOM SCENARIO GENERATION PROCEDURES

The proposed generation procedures shown in Algorithms 1 and 2 are useful when $|\Omega_i|$ is reasonably small since in this case it is possible to analyze all scenarios in Ω_i. Otherwise, it is interesting to generate a smaller set of scenarios $\Omega_i^* \subset \Omega_i$. Analyzing Ω_i^* enables one to infer properties of Ω_i. Deriving Ω_i^* is the focus of this section.

It is clear that a procedure to generate Ω_i^* must be random so that posterior analysis is not biased. A random procedure often makes use of a pseudo-random generator, which is assumed here to be available. More specifically, the algorithms described in this section use the function $\texttt{rand}(a, b)$. Each time $\texttt{rand}(a, b)$ is performed, it returns an integer value k according to a uniform distribution in the interval $[a, b]$. In other words, it is assumed that the probability of choosing a given value k in this interval is $P(k) = 1/(b - a + 1)$.

The problem of generating Ω_i^* at random must satisfy property 4.2 (Validity) and also the following Property:

Property 4.4 (Probabilistic Coverage) *Any subset of Ω_i of size $|\Omega_i^*|$ can be generated.*

Also, the random procedures described in this chapter must ensure termination property. Indeed, one of the derived procedure comply with Property 4.3. Nonetheless, for some random-based procedures it is only possible to guarantee that their expected running time converge to a finite number of steps. In this case, termination requirement is weaker:

Property 4.5 (Probabilistic Termination) *The expected number of steps of the gen-*

eration procedure is bounded.

It can be noted that Property 4.5 is now weaker than Property 4.3. Another difference between the problem stated in the previous section and the one addressed in this section is related to its coverage. Since not all simulation scenarios are to be generated, probabilistic coverage is important to ensure that any subset with the required size can be generated. For example, a naïve deterministic solution to determine Ω_i^* is to consider that it will always contain the first $|\Omega_i^*|$ scenarios in Ω_i so that a sequential generation procedure could be used. However, this solution does not comply with Property 4.4.

Before describing the random-generation procedures, Section 4.2.1 explains the general framework from which they are derived. Then, a very simple procedure is described in Section 4.2.2. This procedure deals with numbers as large as the hyperperiod of the task set, which implies that one must implement long number arithmetics. A second random procedure, which relaxes this need, is presented in Section 4.2.3.

4.2.1 Random Choices to Generate Simulation Scenarios

As explained in Section 3.4, simulation scenarios can be generated by selecting integer arguments from a set of values. This is what has been shown in Algorithm 1, for example. The random generation procedures follow this idea but the selection of such arguments is carried out by random choice instead. Figure 4.1 illustrates a general framework that forms this random selection. The set $\mathcal{U}_i \subset$ represents the arguments that can be used to generate a simulation scenario for task τ_i. The set $\mathcal{U}_i$ must be defined such that any scenario in Ω_i can be generated for the sake of the Probabilistic Coverage property. The goal of the random generation procedure is to select values in $\mathcal{U}_i$ such that a subset $\Omega_i^* \subset \Omega_i$ of scenarios can be generated.

The random probabilistic procedures that will be described work in steps each of which is responsible for choosing a value of $\mathcal{U}_i$. For each chosen value a scenario in Ω_i is generated. If the generated scenario has been generated before, it is discarded and a

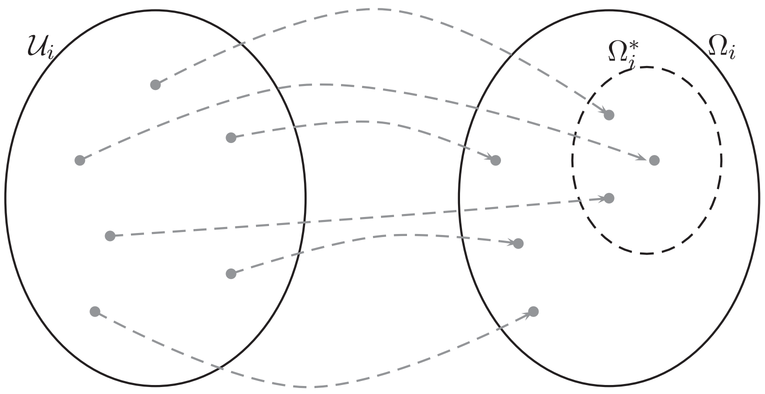


Figure 4.1. Generation of Ω_i^* from $\mathcal{U}_i$.

new value in $\mathcal{U}_i$ must be chosen. The procedure stops when m distinct scenarios in Ω_i has been generated. Algorithm 3 represents this general procedure.

Algorithm 3: General framework to randomly generate a sample of m scenarios for a periodic task set.

```
1 $\Omega_i^* \leftarrow \emptyset$;
2 Carry out some other initialization if needed;
3 repeat
4     Select a number from $\mathcal{U}_i$ at random;
5     Generate a simulation scenario $\mathbf{S}$ from the selected value;
6     if $\mathbf{S} \notin \Omega_i^*$ then
7         $\Omega_i^* \leftarrow \Omega_i^* \cup \mathbf{S}$;
8 until $|\Omega_i^*| = m$ ;
```

One important aspect of any random generation algorithm is its expected running time. Intuitively, it can be seen from Figure 4.1 that depending on the sizes of Ω_i and Ω_i^* the random generation will take more or less steps. Hence, it is useful to derive this time. First, for the sake of notation, the concept of sample size factor is defined:

Definition 4.1 *The sample size factor of $\Omega_i^* \subset \Omega_i$ is defined as*

$$\alpha_i = \frac{|\Omega_i^*|}{|\Omega_i|} \tag{4.1}$$

It is clear that the higher the value of α_i the harder it is to generate Ω_i^*. Now, the following theorem gives the expected number of iterations of any algorithm following the structure of Algorithm 3:

Theorem 4.3 *The expected number of iterations of Algorithm 3 is not greater than*

$$O\left(|\Omega_i| \ln(1-\alpha_i)^{-1}\right) \tag{4.2}$$

Proof The algorithm works in $j = 1, \ldots, |\Omega_i|^*$ steps, each of which is responsible for generating a new element in Ω_i^*. The proof will be by deriving an upper bound on the number of iterations for generating a new scenario in each step j. Since all values in $\mathcal{U}_i$ leads to a valid scenario by Figure 4.1, the probability of generating a valid selection is 1. Hence, it is enough to derive the probability of generating a new scenario in step j. This formulation serves to derive an upper bound on the number of iterations, which will now be done. Define the random variable A_j as follows:

$$A_j \equiv \text{ Generation of a new scenario (not yet in } \Omega_i^*\text{) in step } j$$

Note that in step j, there are $j-1$ scenarios already selected. From this fact, $P(A_j)$ can be computed as

$$P(A_j) = \frac{|\Omega_i| - (j-1)}{|\Omega_i|} \tag{4.3}$$

It can be seen that A_j follows a geometric distribution[1]. From probability theory [54] it follows that its expectation is given by

$$E(A_j) = \frac{|\Omega_i|}{|\Omega_i| - (j-1)} \tag{4.4}$$

Equation (4.4) gives an upper bound on the number of times the **repeat** loop is

[1]Geometric distribution models the number of Bernoulli trials needed to get the first success.

expected to occur for a given value of j. Hence, the bound for $j = 1, 2, \ldots, |\Omega_i^*|$ is

$$\begin{aligned}\sum_{j=1}^{|\Omega_i^*|} \frac{|\Omega_i|}{|\Omega_i| - (j-1)} &= |\Omega_i| \sum_{j=1}^{|\Omega_i^*|} \frac{1}{(|\Omega_i| - j + 1)} \\ &= |\Omega_i| \left(\sum_{j=1}^{|\Omega_i|} (1/j) - \sum_{j=1}^{|\Omega_i| - |\Omega_i^*|} (1/j) \right) \qquad (4.5)\end{aligned}$$

Each sum term in Equation ((4.5)) is an harmonic series. Let H_n, namely the n^{th} harmonic number, be defined as the sum of the first n terms of an harmonic series. Hence, Equation ((4.5)) can be computed as

$$|\Omega_i| \sum_{j=1}^{|\Omega_i^*|} \frac{1}{(|\Omega_i| - j + 1)} = |\Omega_i| (H_{|\Omega_i|} - H_{|\Omega_i| - |\Omega_i^*|}) \qquad (4.6)$$

It is known that $H_n = \ln n + \gamma + O(1/n)$, where γ is Euler's constant [30]. Therefore, assuming $\alpha_i < 1$, Equation ((4.6)) can be approximated as

$$\begin{aligned}|\Omega_i| \sum_{j=1}^{|\Omega_i^*|} \frac{1}{(|\Omega_i| - j + 1)} &\approx |\Omega_i| \ln \left(\frac{|\Omega_i|}{|\Omega_i| - |\Omega_i^*|} \right) \\ &= |\Omega_i| \ln \left(\frac{1}{1 - \alpha_i} \right) \qquad (4.7)\end{aligned}$$

which implies Equation ((4.2)), as required. ▮

It is interesting to observe that the logarithm term in equation (4.7) serves as a reducing factor when $|\Omega_i^*| << |\Omega_i|$. For example, if $|\Omega_i| = 10^8$ and $|\Omega_i^*| = 10^3$, that is $\alpha_i = 10^{-5}$, a random generation procedure in line with the framework of Algorithm 3 is expected to stop with no more than 10^3 iterations. As a rule of thumb, one must not use Algorithm 3 if $\alpha_i \geq 1 - \frac{1}{e}$ since in this case, $\ln(1/(1 - \alpha_i)) \geq 1$ where e is a natural number.

4.2.2 A Simple Random Generation Procedure

Algorithm 4 describes a procedure to generate $m = |\Omega_i^*|$ scenarios at random. In order to select a scenario in Ω_i, an integer k (line 6) is chosen so that $0 \leq kT_i \leq h - T_i$. Note that this range of values ensures that k could assume any multiple of the task periods within $[0, h)$. Then the scenario is generated taking kT_i as argument in $\text{tadd}(\Gamma, \mathbf{S}, kT_i)$. Scenarios already generated are discarded (line 8). If this is the case, another value of k is chosen. The algorithm stops when $j = m$ distinct scenarios are generated. Note the similarities of this algorithm and that of Algorithm 1 and that this random generation procedure is in line with the framework described in Algorithm 3.

Algorithm 4: A simple procedure to randomly generate a sample of m scenarios for a periodic task set.

```
1  $h \leftarrow lcm(T_1, \ldots, T_n)$;
2  $j \leftarrow 1$;
3  $\Omega_i^* \leftarrow \emptyset$;
4  $\mathbf{S} \leftarrow (0, \ldots, 0)$;
5  repeat
6      $k \leftarrow \texttt{rand}(0, (h - T_i)/T_i)$;
7      $\mathbf{S}' \leftarrow \text{tadd}(\Gamma, \mathbf{S}, kT_i)$;
8      if $\mathbf{S}' \notin \Omega_i^*$ then
9          $\Omega_i^* \leftarrow \Omega_i^* \cup \mathbf{S}'$;
10         $j \leftarrow j + 1$;
11 until $j = m$ ;
```

Considering Example 3.2 and task τ_1, k can assume values in the interval $0, 1, \ldots, 5$. In this case, there are 6 possible values of kT_i leading to 6 possible distinct scenarios, as can be seen in Table 4.2.2, which illustrates that probabilistic coverage is ensured.

Since Algorithm 4 is in line with Algorithm 3, the number of iterations is upper bounded by Equation (4.2). Also, the manipulation of simulation scenarios for a task set with n tasks can be done in $O(n \log^2 h)$. Thus, the expected execution cost of Algorithm 4 is upper bounded by $O\left(n \log^2 h |\Omega_i| \log\left(\frac{1}{1-\alpha_i}\right)\right)$. The correctness of the algorithm can be stated as follows:

Table 4.3. Scenarios generated by Algorithm 4 for Example 3.2 considering task τ_1.

kT_i	$\text{tadd}(\Gamma, \mathbf{S}, kT_i)$	kT_i	$\text{tadd}(\Gamma, \mathbf{S}, kT_i)$	kT_i	$\text{tadd}(\Gamma, \mathbf{S}, kT_i)$
0	$(0, 0, 0)$	20	$(20, 15, 20)$	40	$(40, 30, 40)$
10	$(10, 0, 0)$	30	$(30, 30, 20)$	50	$(50, 45, 40)$

Theorem 4.4 *Algorithm 4 generates a subset $\Omega_i^* \subset \Omega_i$ of m simulation scenarios for any set of periodic tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ satisfying properties 4.2, 4.4 and 4.5.*

Proof The Validity property follows directly from line 7 and Lemma 3.1. Also, it can be seen that Algorithm 4 has the same structure of Algorithm 3. Hence, by Theorem 4.3, the algorithm is expected to stop in a finite number of steps, ensuring probabilistic termination by Theorem 4.3. From line 6 it is known that any multiple of T_i within $[0, h)$ can be chosen at random and by Corolary 3.1 these values are enough to generate any scenario in Ω_i, Thus, the Probabilistic Converage property follows. ▮

4.2.3 A Better Random Generation Procedure

It may be clear from Theorem 4.3 that the construction of $\mathcal{U}_i$ plays an important role in the performance of the random generation procedure. In this section another procedure is derived by changing slightly the way $\mathcal{U}_i$ is taken into consideration. Figure 4.2 illustrates the main idea behind the proposed modification. It can be seen that $\mathcal{U}_i$ is divided into three disjunct subsets. A random choice is carried out considering a subset at a time. As the figure indicates, $\mathcal{U}_i$ must be partitioned so that an element is only part of one subset.

Algorithm 5 presents the procedure. Although this algorithm follows the same basic structure as the previous ones, some aspects are worth mentioning. Observe in line 6 that an integer value is chosen between 1 and h^*/T_i, where h^* is the size of the partitions of $\mathcal{U}_i$. Its value must be large enough to ensure the Probabilistic Coverage property. Otherwise, the algorithm will stop before taking the whole spectrum of $\mathcal{U}_i$. In other words, $h^* \geq h/m$,

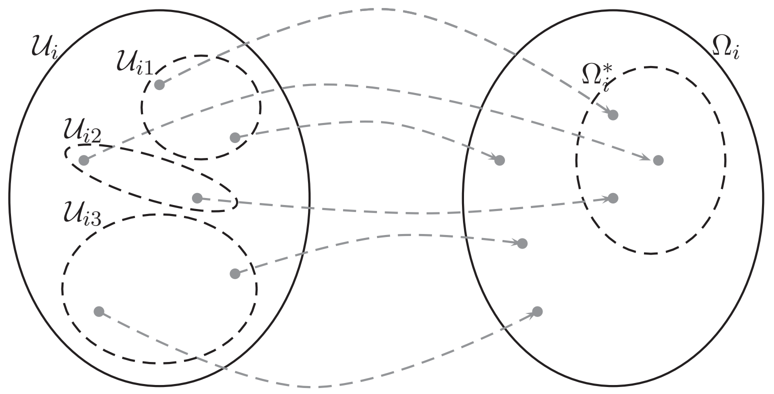


Figure 4.2. Partitioning $\mathcal{U}$ into disjunct subsets.

where m is the desired number of simulation scenarios in $|\Omega_i^*|$. Another aspect is the operation in line 10. This operation has the effect of advancing time so that the domain for random choices are kept bounded within the chosen partition size. Both the time advance and the time shift operations (lines 10 - 11), are used to reduce the complexity due to large number arithmetics. This is a positive side effect of the strategy described in this algorithm.

Algorithm 5: A better procedure to randomly generate a sample of m scenarios for a periodic task set assuming $m < h/T_i$.

```
1  $\Omega^* \leftarrow \emptyset$;
2  $\mathbf{S} \leftarrow (0, \ldots, 0)$;
3  $h \leftarrow \mathrm{lcm}(T_1, \ldots, T_n)$;
4  $h* = \left\lfloor \frac{h}{m} \right\rfloor$;
5  repeat
6      $k \leftarrow \texttt{rand}\left(0, \left\lfloor \frac{h^* - T_i}{T_i} \right\rfloor\right)$;
7      $\mathbf{S}' \leftarrow \mathrm{tadd}(\Gamma, \mathbf{S}, kT_i)$;
8      $\Omega_i^* \leftarrow \Omega_i^* \cup \mathbf{S}'$;
9      $j \leftarrow j + 1$;
10     $\mathbf{S} \leftarrow \mathrm{tadd}(\Gamma, \mathbf{S}, h^*)$;
11     $\mathbf{S} \leftarrow \mathrm{tshift}(\mathbf{S}, -S_1)$;
12 until $(j = m)$ ;
```

Consider again Example 3.2 and task τ_1 as illustration and define $h^* = 20$. This means that the algorithm chooses a value of $kT_1 \in \{0, 10, 20\}$ in each of its steps. In the first step, as $\mathbf{S} = (0, 0, 0)$, there are two possible scenarios to be generated. In the second step, though, only two different scenarios can be generated. Table 4.4 gives the possible scenarios that could be generated by the algorithm in each of its steps. Note that in order to ensure the Probabilistic Coverage property, there must be at least three steps. The demonstration that Algorithm 5 follows the required properties is defined in the following theorem.

Theorem 4.5 *Algorithm 5 generates a subset $\Omega_i^* \subset \Omega_i$ of m simulation scenarios of a periodic task $\tau_i \in \Gamma = \{\tau_1, \ldots, \tau_n\}$ satisfying properties 4.2, 4.4 and 4.5.*

Proof

Validity. The Validity property follows directly from line 7 and Lemma 3.1.

Probabilistic Coverage. From lines 6-7 it is known that any multiple of a task period within $[0, h^*)$ can be chosen at random. There are m intervals with size h^* (line 4). Note that **repeat-until** loop and line 10 assures that all intervals are visited. Hence any multiple of T_i within [0,h) can be chosen at random. According to Corolary 3.1 these values are enough to generate any scenario in Ω_i. Thus, the Probabilistic Coverage Property follows.

Termination. Note that the procedure divides h in $m < h/T_i$ distinct subsets, where m is the number of scenarios that must be generated. In each subset a scenario will always be generated. Since there are m subsets, there will be m generated scenarios. Hence, Algorithm 5 terminates with probability 1 in m steps.

∎

In spite of calculating the hyperperiod, Algorithm 5 generates new scenarios based on the values of h^* (line 4) instead of using h. As a consequence it reduces the complexity due to large numbers arithmetics, which is enforced by the use of **time-shift** operation (line 11). Moreover, Algorithm 5 divides the hyperperiod in m subintervals and each

Table 4.4. Illustration of Algorithm 5 for the illustrative example

step	$\mathbf{S}$	k	$\mathbf{S}'$	$\mathbf{S}' \equiv$
1st	$(0,0,0)$	0	$(0,0,0)$	$(0,0,0)$
		1	$(10,0,0)$	$(10,0,0)$
2nd	$(0,-5,0)$	0	$(0,-5,0)$	$(20,15,20)$
		1	$(10,10,0)$	$(30,30,20)$
3rd	$(0,5,0)$	0	$(0,5,0)$	$(40,45,40)$
		1	$(10,5,0)$	$(50,45,40)$

one of them generates a scenario in $O(n \log^2 h^*)$ (line 7). Hence, the running time of Algorithm 5 is $O(mn \log^2 h^*)$.

4.3 SUMMARY

This chapter presented four procedures to generate *simulation scenarios*. Two of them are useful when the total number of simulation scenarios for a given task τ_i is reasonably small. They generate the whole set of such scenarios. However, in general, complex task sets have a very large set of simulation scenarios. To deal with these cases, two random generation procedures were derived. Table 4.5 summarizes the characteristicis of the derived procedures.

Choosing an adequate algorithm to generate simulation scenarios poses as an important issue when it comes to simulating a specific time window of a system, instead of considering the whole hyperperiod. The following chapter will presents the Simulation Engine, which uses simulation scenarios to define the simulation time interval and computes the fault resilience metric.

Table 4.5. Summary of simulation scenarios generation procedures

Generation Procedure	Time Complexity	Characteristics
Algorithm 1	$O\left(\frac{nh}{T_i}\log^2 h\right)$	Generates Ω_i for a given task τ_i; Based on hyperperiod; Generates a scenario in every iteration. Arithmetic of large numbers.
Algorithm 2	$O\left(\frac{nh}{T_i}\right)$	Generates Ω_i for a given task τ_i; Not based on hyperperiod; Generates a scenario in every iteration.
Algorithm 4	$O\left(\frac{nh}{T_i}\log\left(\frac{h}{h-mT_i}\right)\log^2 h\right)$	Generates a subset Ω_i^* for a given task τ_i; Based on the hyperperiod; Generates a scenario in every iteration. Arithmetic of large numbers.
Algorithm 5	$O(mn\log^2 h^*)$	Generates a subset Ω_i^* for a given task τ_i; Based on the hyperperiod; Generates a scenario in every iteration. Deals with numbers bounded within a few values.

CHAPTER 5

SIMULATION ENGINE

Chapter 4 presented some procedures to generate a set of simulation scenarios for a given task τ_i. Based on each scenario $\mathbf{S}$ the *simulation engine* determines a specific time window and then simulates the system to compute the fault resilience of the system regarding $\mathbf{S}$.

Indeed, in order to determine the system behavior in a specific time window it is necessary to evaluate if the execution of previous time windows may interfere in the one that is being simulated. In other words, in order to carry out this simulation, the impact of possible pending execution from previous scenarios in the one that is being simulated must be considered. This procedure is addressed here as the *backlog computation*. Also, the error generator procedure must generate errors so that the fault resilience is computed considering the minimum number of errors and this is the second issue addressed in this chapter.

The outline of the proposed simulation-based approach is given in Section 5.1. The backlog effect on the simulation is analyzed in Section 5.2. Section 5.3 details both the error generator and the system fault resilience. This chapter finishes with some preliminar results obtained from the error generator algorithm, which are presented in Section 5.4.

5.1 OUTLINE

Assume that one wishes to evaluate the effort for a task τ_i in $\Gamma = \{\tau_1, \ldots, \tau_n\}$, considering a specific simulation scenario $\mathbf{S} = (S_1, \ldots, S_n)$. The job of this task, released at S_i, namely J_i, is called hereafter the analyzed job. Simulation will be taken until J_i misses its deadline, making the system unschedulable. Figure 5.1 sketches the simulation process and will be used for illustration purposes. Scenario $\mathbf{S}$ and a previous scenario $\mathbf{S}'$ are indicated in the gray area of the figure. Release times are indicated by the vertical solid arrows. The first release time of jobs in **S** whose priorities are at least $p(J_i)$ is denoted $r \leq S_i$ in the figure. These jobs must be considered when analyzing the effects of errors in the execution of J_i, since they may cause some interference in such a job.

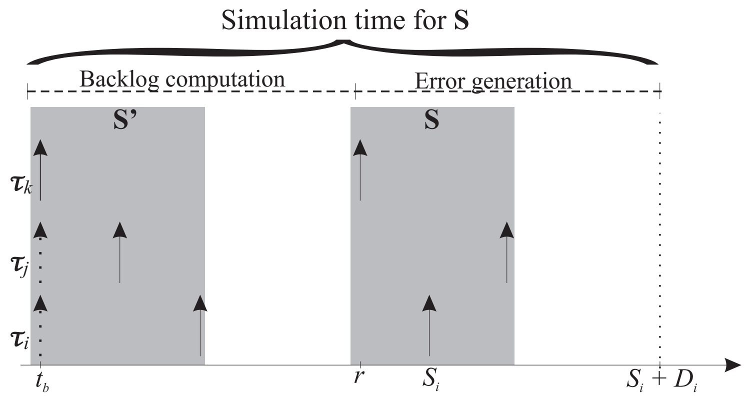


Figure 5.1. Two-step simulation procedure used by the Simulation Engine

The simulation of the system regarding **S** involves two problems: (a) determining the execution backlog at r, which is related to jobs released before r; and (b) generating the minimum number of errors from r onwards so that the analyzed job misses its deadline. Nonetheless, exact solutions to problems (a) and (b) may be computationally too expensive for two main reasons: first, analyzing the impact of higher priority jobs already released on the analyzed job requires backtracking in time. Second, generating the minimum number of errors is an optimization problem. Thus, our approach to solving them is to derive an upper bound for (a) and a lower bound for (b) so that the effort of the fault generator is not overestimated. The simulation procedure has two steps, as illustrated in Figure 5.1, and is explained in Sections 5.2 and 5.3, respectively.

5.2 BACKLOG COMPUTATION

The approach to estimating an upper bound on the backlog at r is based on (a) going back to a previous scenario $\mathbf{S}'$ and (b) forcing the release time of all tasks in Γ be at time $t_b = \min(\mathbf{S}')$. Then, the remainder task execution time after simulating the execution of the system within $[t_b, r)$ should give the desired upper bound. It is worth mentioning that any previous simulation scenario $\mathbf{S}'$ could be considered. For example, one could choose to backtrack from $\mathbf{S}$ until the time origin at $t = 0$, since this choice would lead to compute the worst-case for backlog. However, in worst case it would be necessary to consider the whole hyperperiod, which is not desired. Indeed, ideally $\mathbf{S}'$ should be a scenario which gives a good trade-off between simulation time and backlog estimation.

In order to evaluate which backtrack would give the ideal t_b value, 5 periodic task sets, composed of six tasks each, were simulated. Such task sets were randomly generated, considering variable values for both task periods $\mathbf{T}$ and execution cost $\mathbf{C}$. In this simulation, for each task τ_i we chose a simulation scenario $\mathbf{S}$ and backtracked to a previous scenario $\mathbf{S}'$, where $\mathbf{S}' = \text{tsub}(\Gamma, \mathbf{S}, kT_i), k = 0, 1, \ldots, \frac{h-T_i}{T_i}$. The mean backlog computed for each task τ_i is presented in Table 5.1, where k indicates the number of scenarios before $\mathbf{S}$ that were considered to compute the backlog.

Table 5.1. Summary of the mean backlog considering $\mathbf{S}' = \text{tsub}(\Gamma, \mathbf{S}, kT_i)$

i	k							
	50	40	30	20	15	10	5	1
1	0.0100	0.0103	0.0149	0.0161	0.0220	0.0243	0.0253	0.0260
2	0.0103	0.0106	0.0139	0.0145	0.0200	0.0223	0.0231	0.0239
3	0.0101	0.0103	0.0132	0.0150	0.0203	0.0219	0.0230	0.0241
4	0.0105	0.0109	0.0127	0.0153	0.0211	0.0224	0.0245	0.0253
5	0.0105	0.0108	0.0137	0.0151	0.0207	0.0233	0.0251	0.0266
6	0.0110	0.0113	0.0149	0.0157	0.0220	0.0239	0.0260	0.0269

Notice that the mean backlog values presented a small variability if we consider a backtrack of at most 15 scenarios. Actually, such values begin to change when $k \geq 20$. Also, notice that the greatest backlog values were computed for the closest simulation scenarios. This was an expected behavior since the closer $\mathbf{S}$ is from $\mathbf{S}'$, the smaller the time available to compute the extra workload. Based on that, the approach used to compute the backlog, in the context of this work, is going back to the closest scenario. In this case, $\mathbf{S}'$ and t_b are respectively given by:

$$\mathbf{S}' = \text{tsub}(\mathbf{S}, \Gamma, kT_i), k > 0 \qquad t_b = \min(\mathbf{S}')$$

Once t_b is computed, the simulation starts executing the jobs released in $[t_b, r)$ but with the fault generator deactivated. Since some jobs may be artificially released at t_b, as illustrated by the dotted-arrowed lines in Figure 5.1, there may be an execution overload in $[t_b, r)$ which are generated for the purpose of backlog estimation only. This is illustrated in Figure 5.2. For example, let $\mathbf{S} = (50, 45, 40)$ and $\mathbf{S}' = (30, 30, 40)$. Artificially, a job of τ_3 is considered to be released at $t = 30$. Also, observe that the job of τ_2 released at time instant 30 does not complete its execution during the time interval $[30, 40)$ and thus its pending execution time represents a backlog for $\mathbf{S}$.

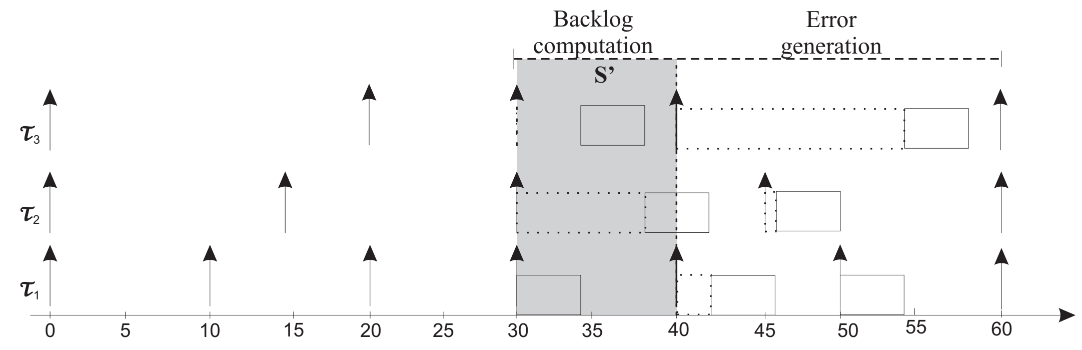


Figure 5.2. Illustrative example of backlog computation

In order to reduce this artificial overload, the jobs that miss their deadlines in $[t_b, r)$ are executed until their deadlines, time at which they are discarded. This strategy can be observed in Figure 5.3. In this case, $\mathbf{S} = (30, 30, 20)$ and $\mathbf{S}' = (20, 15, 20)$. Observe that at time $t = 20$, a backlog job, released at time $t = 15$ is discarded. The jobs simulated during $[t_b, r)$ are called *backlog jobs*. Since we assume that the system is schedulable in fault-free scenarios, discarding backlog jobs in this way is safe and reduces the pessimism of the simulation-based analysis.

5.3 ERROR GENERATOR

The simulation during $[r, S_i + D_i)$ is carried out with the fault generator active. The strategy is to generate errors in the job which causes the highest interference in the analyzed job. As the goal is to estimate a lower bound on the minimum number of

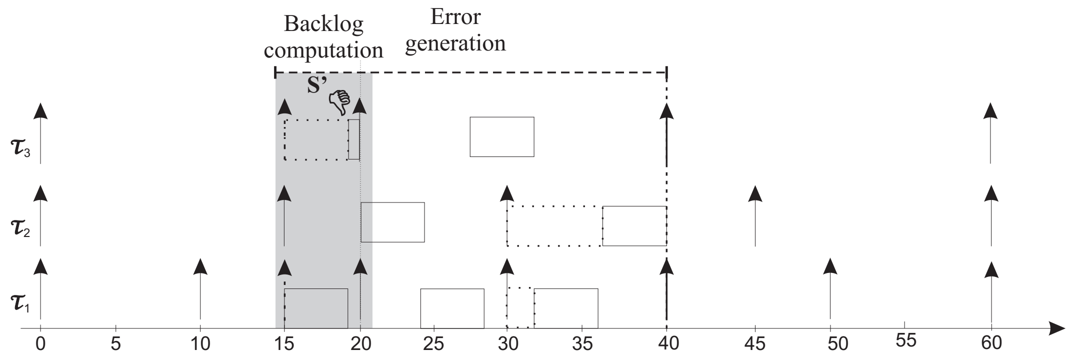


Figure 5.3. Illustrative example of a backlog job being discarded

generated errors that make the analyzed job miss its deadline, the faulty jobs are allowed to execute beyond its deadline. In other words, the optimization problem of determining which jobs fail during simulation is circumvented. According to this approach the found number of errors is guaranteed to be not overestimated but can be underestimated.

Consider a time interval $[r, t)$, $r < t$, during which the fault generator is active when simulating a given scenario **S**. Assume that the analyzed job J_i is active in this interval. We define the predicate $\texttt{active}(r, t, J_i)$, which is true whenever J_i has been released before t and has not finished by r. Otherwise, the predicate evaluates false. By the scheduling policy, the execution of J_i takes place at priority level $p(J_i)$. Hence, the jobs active in $[r, t)$ that may interfere in the execution of the analyzed job or in its recovery are executed or have their recovery actions executed at priority at least $p(J_i)$. We represent this set more formally as

$$hp_i^S(r,t) \overset{\text{def}}{=} \{J_j | (p(J_j) \geq p(J_i) \vee \bar{p}(J_j) \geq p(J_i)) \wedge (\texttt{active}(r, t, J_j)\}$$

As mentioned before, when simulating a scenario **S** for a given analyzed job J_i, the fault generator must not let J_i meet its deadline. Hence, every time t at which J_i would successfully finish its execution (i.e $t \leq S_i + D_i$), an error must be generated in a job $J_j \in hp_i^S(r, t)$. The recovery of this faulty job must be such that it maximizes the interference in the execution of J_i. Notice that jobs released in $[r, S_i)$ may satisfy the maximization criterion. In order to compute this kind of interference, we define the concept of *interference distance*:

Definition 5.1 *Let* $\mathbf{S} = (S_1, \ldots, S_n)$ *be a simulation scenario for a given periodic task set. Consider a time interval* $[r, t)$ *and a job* J_i*, where* $\min(\mathbf{S}) \leq r < t \leq S_i + D_i$*. For any job* $J_j \in hp_i^S(r, t)$*, denote* e_j *the finishing time of* J_j *when no errors take place and define* $C_k(t)$ *the pending worst-case execution cost at time* t *of any* $J_k \in hp_i^S(r, t)$*. If* J_k *has not executed by* t*,* $C_k(t) = C_k$*. If it has finished by* t*,* $C_k(t) = 0$*. Otherwise,* $C_k(t)$ *equals* C_k *minus what has been executed of* J_k *by* t*. The interference distance,* $\Delta_{i,j}^S$*, of job* J_j *over job* J_i *in* $\mathbf{S}$ *is given by:*

$$\Delta_{i,j}^S \overset{\text{def}}{=} \max\{S_i - e_j - \omega_i(e_j), 0\} \tag{5.1}$$

where

$$\omega_i(t) \overset{\text{def}}{=} \sum_{J_k \in hp_i^S(r,t)} C_k(t) \tag{5.2}$$

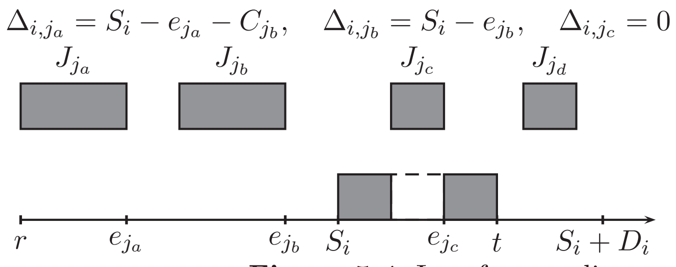
Figure 5.4 illustrates the concept stated in Definition 5.1. Time t is a possible successful finishing time of J_i. At this moment, the fault generator must generate an error in some job in $hp_i^S(r, t)$. There are three possibilities in this example, denoted in the figure as J_{j_a}, J_{j_b} and J_{j_c}. Note that J_{j_d} is only active after t and so it is not considered as an option. Also, note that the interference distance $\Delta_{i,j_c}^{\mathbf{S}} = 0$ and so any error in J_{j_c} would cause an extra interference in the execution of J_i. This is not true for jobs J_{j_a} and J_{j_b}. Indeed, J_i would suffer interference of these jobs due to errors only if their recovery times are greater than their interference distances.

In the example no faults are considered. Now consider a general case where the fault generator is to generate an additional error at time t and $f_i^{\mathbf{S}}$ have already been generated. Let the execution time of recovery actions associated to all these $f_i^{\mathbf{S}}$ errors be $\bar{C}$ time units. Thus, the recovery action associated to all $f_i^{\mathbf{S}} + 1$ errors is either

$$\max_{J_j \in hp_i^S(r,S_i)} \left\{(f_i^{\mathbf{S}} + 1)\bar{C}_j - \Delta_{i,j}^{\mathbf{S}}\right\} \tag{5.3}$$

or

$$\bar{C} + \max_{J_j \in hp_i^S(S_i,t)} (\bar{C}_j) \tag{5.4}$$



$\Delta_{i,j_a} = S_i - e_{j_a} - C_{j_b}$, $\Delta_{i,j_b} = S_i - e_{j_b}$, $\Delta_{i,j_c} = 0$
J_{j_a}
J_{j_b}
J_{j_c}
J_{j_d}
r
e_{j_a}
e_{j_b}
S_i
e_{j_c}
t
$S_i + D_i$

Figure 5.4. Interference distance.

The maximum of Equations (5.3) and (5.4) gives the desired lower bound on the number of generated errors whenever $f_i^{\mathbf{S}} + 1$ errors make J_i miss its deadline. It is important to emphasize that this strategy can be used because we are considering that the execution of faulty jobs is not time-bounded, i.e. a chosen faulty job J_j is assumed to execute beyond its deadline $S_j + D_j$. This is done for the sake of analysis only and does not imply that we are restricting the system task model. It is clear that since J_j has the highest interference in the execution of J_i according to this assumption, any other combination of faulty jobs cannot interference more than what is computed in Equations (5.3) and (5.4). Therefore, we are conservatively determining the fault generator effort as mentioned before. Less pessimistic approaches to choosing faulty jobs involve solving optimization problems, which will be considered in future work. For now, we are more concerned with validating the fault resilience analysis as a whole.

5.3.1 Simulation Procedure

Algorithm 6 implements the simulation engine. It receives as input parameters a task set Γ, one of its simulation scenarios $\mathbf{S}$, and one task to be analyzed regarding $\mathbf{S}$, whose job is released at S_i. The simulation interval $[t_b, t')$ is set in lines 1-3. Variables $\bar{C}$ and $f_i^{\mathbf{S}}$ in the algorithm stores the sum of recovery times of faulty jobs and the number of errors, respectively. The final value of $f_i^{\mathbf{S}}$ is the generated number of errors by the fault

Algorithm 6: Simulation engine

```
1  $t' \leftarrow S_i + D_i$; $r \leftarrow \min_{J_k \in \mathrm{hp}_i^S}(S_k)$;
2  $\mathbf{S}' = \mathrm{tsub}(\Gamma, \mathbf{S}, \min(\mathbf{T}))$;
3  $t_b \leftarrow \min_{J_k \in \mathrm{hp}_i^{S'}}(S'_k)$; $t \leftarrow t_b$;
4  foreach $J_k \in \mathrm{hp}_i^S(t_b, t')$ do
5      enqueue$(k, C_k, p(J_k))$;
6  enqueue$(0, t - t', p(J_i) - 1)$;                          /* a dummy job */;
7  $f_i^S \leftarrow 0$; $\bar{C} \leftarrow 0$;
8  while $(t \leq t')$ do
9      $(k, C, p) \leftarrow$ dequeue$(t)$;
10     $s \leftarrow$ nextJob$(t, p)$;
11     if $t + C \leq s$ then                                  /* $J_k$ finishes */
12         $t \leftarrow t + C$;
13         if $t < r$ then                                     /* backlog job */
14             if $S_k + D_k > t + C$ then                     /* overload?  */
15                 $t \leftarrow S_k + D_k$;
16         else                                                /* fault generator active */
17             if $j = i \wedge t \leq S_i + D_i$ then
18                 $f_i^S \leftarrow f_i^S + 1$;
19                 $x \leftarrow \max_{J_j \in \mathrm{hp}_i^S(r, S_i)}(f_i^S \bar{C}_j - \Delta_{i,j}^S)$;
20                 $y \leftarrow \max_{J_j \in \mathrm{hp}_i^S(S_i, t)}(\bar{C}_j)$;
21                 if $x > \bar{C} + y$ then
22                     enqueue$(i, x - \bar{C}, p(J_i))$;
23                     $\bar{C} \leftarrow x$;
24                 else
25                     enqueue$(i, y, p(J_i))$;
26                     $\bar{C} \leftarrow \bar{C} + y$;
27
28     else                                                    /* $J_k$ is preempted */
29         enqueue$(k, C - (s - t), p)$;
30         $t \leftarrow s$;
31
```

generator that make J_i miss its deadline, meaning that scenario $\mathbf{S}$ is resilient to at least $f_i^{\mathbf{S}} - 1$ errors.

Initially, all jobs in the simulation interval are enqueued according to their priorities and release times (lines 4-5). This is carried out by the function `enqueue`. A dummy job is also enqueued at priority level $p(J_i) - 1$ (line 6). This job is executed in background during the simulation. Since time advances in the algorithm only when some job is executed, the dummy job is used for the purpose of advancing time during idle intervals.

Any job is dispatched to execution at time t as follows. The highest priority ready job at t is dequeued (line 9). The `dequeue` function returns the job identifier (k), its execution time (C) and its priority (p). Then the `nexJob` function is called. This function returns the next release time of the job with priority at least p whose release time is greater than t. If $t + C > s$, the dispatched job J_k is executed until time s when a preemption occurs. Otherwise, there are three situations to be checked. If J_k is a backlog job, it is executed until either time $t + C$ or time $S_k + D_k$. In the former case, time is simply advanced to $t + C$. In the latter case, J_k misses its deadline at time $S_k + D_k$ and is discarded (lines 14-15). Finally, if $J_k = J_i$ and J_i meets its deadline, an additional error is generated in the job which maximizes either Equation (5.3) or Equation (5.4), as explained earlier.

Note that the recovery time of the faulty job is added to C_i. This avoids possible backtracking to execute the recovery of J_j, simplifying the simulation. Also, preemption is simulated (lines 28-30) simply by enqueuing the remainder execution time ($C - s + t$) of the current job and advancing time to s. When `dequeue` is called at time s, the highest priority job at s is selected.

The correctness of the simulation engine is stated by the following theorem.

Theorem 5.1 *Consider a periodic task set* $\Gamma = \{\tau_1, \ldots, \tau_n\}$ *and a given simulation scenario* $\mathbf{S} = (S_i \ldots, S_n)$ *of* Γ. *Algorithm 6 finishes its execution returning a lower bound on the number of errors necessary to make the analyzed job released at* S_i *miss its deadline.*

Proof [Proof sketch] It is not difficult to see that the algorithm halts since in each of its steps in the iterative loop time advances in lines 12, 15 or 30 and so eventually $t > t'$.

Assume that $f < f_i^{\mathbf{S}}$ errors would make J_i miss its deadline in an actual schedule. Since the value of $f_i^{\mathbf{S}}$ is increased one unit at a time throughout the execution of the algorithm, consider a moment t when $f_i^{\mathbf{S}} = f$ and J_i finishes successfully in the simulation. Then either an error is generated in some job in $\text{hp}_i^S(S_i, t)$ or all errors take place in some job in $\text{hp}_i^S(r, S_i)$. Since the backlog at r is not underestimated, these sets of jobs contain at least the same jobs which would be active in the corresponding intervals $[r, S_i)$ and $[S_i, t)$ in an actual schedule. Since the choice of faulty jobs maximizes the interference in the execution of J_i due to error recovery, J_i would meet its deadline in the actual schedule if no more than f errors were taken into consideration, a contradiction. ∎

5.3.2 Illustrative Example

To make things clear, consider $\mathbf{S} = (50, 45, 40)$ a simulation scenario for Example 3.2 scheduled according to EDF. In this scenario let us analyze the job J_3 released at $t = 20$. For such a scenario, $\mathbf{S}' = (\text{tsub}(S, \Gamma, \min(\mathbf{T})) = (40, 30, 40)$ and the simulation time interval is $[30; 60)$. Figure 5.5 shows the whole scheduling for $\mathbf{S}$, including the backlog computation and error generation.

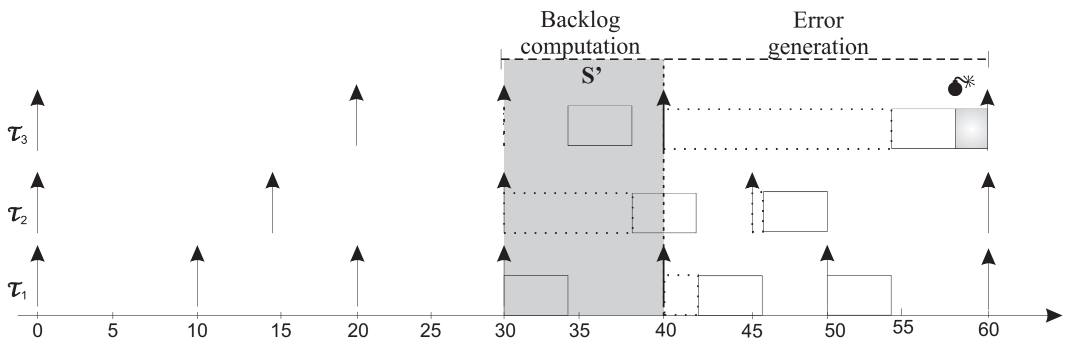


Figure 5.5. Illustrative simulation for Example 3.2

Observe that, in this case, system simulation starts at $t_b = 30$. At this time, an instance of τ_3 is artificially included (dotted arrow in Figure 5.5). As mentioned before, this is done for deriving an upper bound on the backlog. Notice that at $t = 40$, a backlog job from previous scenario (from task τ_2) is still executing. It is worth mentioning that in the actual schedule shown in Figure 3.5 the backlog is null.

At time instant $t = 40$ the analyzed job is released and starts executing at $t = 54$ due to the interference of higher priority jobs. When the error generator realizes that this job is going to finish its execution successfully, it generates an error so that this job miss its deadline, making the system unschedulable. The choice of the faulty job is done so that the error generator effort is minimized. Hence, any active job in $[40, 60)$ can be chosen as the faulty job provided that it causes the highest interference in the analyzed job. It is important to notice that during the time interval $[40, 60)$, if any other job released at this time interval misses its deadline the system is still considered schedulable. Only the timeliness of the analyzed job (J_3) is observed. According to Definition 3.2, the effort value for $\mathbf{S}$ is given by $\mathbf{E}_3 = \frac{f_3^{\mathbf{S}}}{D_3} = \frac{1}{20} = 0.05$.

In the following section we present some simulation results showing how the derived fault resilience metric can be used to compare different systems.

5.4 PRELIMINARY RESULTS

In this section we give some direction of the proposed analysis using an example task set composed of 10 tasks. Both EDF and RM were considered. The example was generated as follows. The task execution times were fixed and the task periods were randomly generated so that processor utilization was in $[0.6; 0.8)$. The recovery model considered was task re-execution and so $\bar{\mathbf{C}} = \mathbf{C}$:

Example 5.1 *Consider* Γ *a periodic task set with* 10 *tasks and* $\mathbf{C} = (1, \ldots, 1)$, $\mathbf{T} = (3, 11, 14, 15, 19, 19, 28, 33, 35, 44)$, $\mathbf{D} = \mathbf{T}$ *and* $\bar{\mathbf{C}} = \mathbf{C}$.

The hyperperiod for the analyzed task set is $h = 87,780$, which leads to $\sum_{i=1}^{10} \Omega_i = 68,898$. The mean effort $\bar{\mathbf{E}}_\mathbf{i}$ for both RM and EDF is presented in Table 5.2.

Table 5.2. Mean effort $\bar{E}_i$ for Γ

	$\bar{\mathbf{E}}_i$									
i	1	2	3	4	5	6	7	8	9	10
RM	1.000	0.614	0.432	0.337	0.294	0.262	0.247	0.206	0.173	0.162
EDF	0.999	0.658	0.507	0.406	0.371	0.371	0.332	0.291	0.282	0.256

As can be seen in the table, EDF has a better overall performance in terms of fault resilience than RM. Although this behavior was expected due to the optimality of EDF in terms of schedulability, it is important to emphasize that now the difference is being measured. It is worth mentioning that τ_1 has the same fault resilience for both schedulers. Indeed, $\bar{\mathbf{E}}_\mathbf{i} = 1$ for both EDF and RM, although for EDF a small variability ($< 10^{-3}$) was present. Since no other task interferes in the execution of τ_1 according to RM and only in a few simulation scenarios there are other tasks with priority greater than $p(J_1)$ by EDF, this behavior was also expected. On the other hand, for all other tasks, EDF is visibly superior to RM in terms of fault resilience. Obviously, we are not considering here problems such as possible overloads caused by the admission of recovery actions, which could make EDF degrade. Nonetheless, the goal of the analysis is to point out to what extent the system support errors and is not on evaluating overload conditions.

5.5 SUMMARY

This chapter presented an approach to infering the a system resilience considering simulation scenarios. Two main problems were addressed: the first one deals with the impact of pending execution from previous scenarios in the one that is being simulated; the error generator procedure using the described approach, it was illustrated that EDF is superior to RM from fault resilience perspective. The following chapter will present a more systematic analysis of the simulation data produced by the simulation engine.

CHAPTER 6

STATISTICAL ANALYSIS

The approach presented in Chapter 5 illustrated how to compute the fault resilience for a given task based on simulation scenarios. Indeed, when all simulation scenarios are available, the simulation engine can compute the effort for each task and the fault resilience of a given system can be determined based on the desired parameters, such as the mean effort, median or percentile values. However, there may be situations in which simulating all scenarios may be too time consuming. In this case, the fault resilience effort of a given task, for example, can be determined based on a sample of simulation scenarios, which requires applying statistical inference.

This chapter focus on describing two statistical inference methods, which can be used to determine the confidence interval for a specific parameter based on a sample of such values. When the chosen parameter is the mean, traditional inference, which is detailed in Section 6.1, can be safely used to determine the confidence interval. However, if the chosen parameter is different from the mean, bootstrap technique, presented in Section 6.2, is recommended. In this chapter we apply both methods for computing the 95% confidence interval for the chosen parameter, which is shown through illustrative examples in Sections 6.1 and 6.2. Simulation results are presented and discussed in Section 6.3.

6.1 TRADITIONAL INFERENCE

Statistical inference methods deal with the problem of drawing conclusions for a population based on sample data. More precisely, it deals with two problems: (a) testing hypothesis about a specific parameter, issue not discussed here, and (b) estimating a specific parameter through sample data, the focus of this work. Here, we use interval estimation, which aims at determining a $100(1-\alpha)\%$ confidence interval for a specific parameter, where $1-\alpha$ is the confidence coefficient. Parameter represents a population attribute that one wishes to study and the $100(1-\alpha)\%$ confidence interval represents an estimated range of values, which is likely to include the population parameter.

Traditional inference techniques, which are based on the Central Limit Theorem [54, 40], have been extensively used to determine the confidence interval when the parameter is the mean. In order to illustrate how to use such a kind of technique, consider an illustrative example which was presented in Section 5.4 and repeated here.

Example 6.1 *Consider* Γ *a periodic task set with* 10 *tasks and* $\mathbf{C} = (1,\ldots,1)$, $\mathbf{T} = (3, 11, 14, 15, 19, 19, 28, 33, 35, 44)$, $\mathbf{D} = \mathbf{T}$ *and* $\bar{\mathbf{C}} = \mathbf{C}$.

Assume that the parameter one wishes to study is the mean effort $\bar{\mathbf{E}}_\mathbf{i}$ for each task $\tau_i \in \Gamma$. In order to construct a confidence interval for $\bar{\mathbf{E}}_i$, we need first to calculate the sample size $|\Omega_i^*|$ for each task τ_i. To do so we need to (a) determine an acceptable *sample error* for each task, which is denoted by $|\bar{\mathbf{E}}_i^* - \bar{\mathbf{E}}_i|$, where $\bar{\mathbf{E}}_i^*$ and $\bar{\mathbf{E}}_i$ stand for the mean effort related to the sample Ω_i^* and to the population Ω_i, respectively; (b) fix the population standard deviation σ_i and (c) determine the coefficient confidence $1-\alpha$. For this case, we have set $|\bar{\mathbf{E}}_i^* - \bar{\mathbf{E}}_i| = 5\text{x}10^{-3}$ and $\alpha = 5\%$. The population standard deviation σ_i was assumed to be known, and the sample size $|\Omega_i^*|$ for each task $\tau_i \in \Gamma$ was computed according to Equation(6.1) [54, 40].

$$|\Omega_i^*| = \frac{1.96\sigma_i}{|\bar{\mathbf{E}}_i^* - \bar{\mathbf{E}}_i|} \tag{6.1}$$

Note that if σ_i is not known, it can be estimated from a pilot sample [54]. The sample size computed for each task $\tau_i \in \Gamma$ are shown in Table 6.1.

Table 6.1. Sample size estimation for Example 6.1

i	1	2	3	4	5	6	7	8	9	10
$\lvert\Omega_i^*\rvert$	56	589	1135	851	683	713	389	326	248	219

A random generation procedure was used to generate the random samples of simulation scenarios for each task $\tau_i \in \Gamma$ based on the sample size $|\Omega_i^*|$. Although bothe Algorithms 4 and 5 can be used, we used Algorithm 5 since it presents a better performance. Each sample of simulation scenarios Ω_i^* was used as input for the simulation engine, so that the mean effort $\bar{\mathbf{E}}_i^*$ was computed. Such values are shown in Table 6.2.

Table 6.2. $\bar{E}_i$ with 95% of confidence

	RM			EDF		
i	$\bar{\mathbf{E}}_i^*$	CI_i	$\bar{\mathbf{E}}_i$	$\bar{\mathbf{E}}_i^*$	CI_i	$\bar{\mathbf{E}}_i$
1	1.000	[1.000,1.000]	1.000	1.000	[1.000,1.000]	0.999
2	0.614	[0.505,0.723]	0.614	0.658	[0.536,0.780]	0.661
3	0.432	[0.284,0.579]	0.433	0.507	[0.325,0.688]	0.506
4	0.337	[0.220,0.455]	0.338	0.406	[0.261,0.550]	0.407
5	0.294	[0.196,0.391]	0.295	0.371	[0.249,0.493]	0.375
6	0.262	[0.155,0.369]	0.264	0.371	[0.237,0.505]	0.373
7	0.247	[0.131,0.362]	0.247	0.332	[0.233,0.431]	0.332
8	0.206	[0.115,0.297]	0.207	0.291	[0.198,0.384]	0.291
9	0.173	[0.123,0.223]	0.174	0.286	[0.200,0.364]	0.280
10	0.162	[0.112,0.211]	0.161	0.256	[0.183,0.329]	0.256

The confidence interval CI_i for the parameter $\bar{\mathbf{E}}_i$ was also computed considering a confidence of 95%. For this example, the mean effort $\bar{\mathbf{E}}_i$ for all simulation scenarios Ω_i was known and is also presented in the table. Observe that most effort values obtained for EDF are greater than the ones obtained for RM, except for task τ_1 in which such values are the same. Note that this is the same result presented in Table 5.2, illustrating the effectiveness of statistical estimation.

Although such a method to calculate a confidence interval has been extensively used in statistical literature, it may present some shortcomings if one is interested in a population parameter which is different from the mean. In fact, in this case computing the $(1-\alpha)\%$ confidence interval for a parameter cannot be done straightaway. The following section presents a computer-based inference technique which can be used to infer population parameters, which are different from the mean.

6.2 BOOTSTRAP

Bootstrap is a computer-based method for statistical inference, which uses sample data to estimate sample distribution without the need of carrying any model assumption. Quoting Politis [46], the idea behind bootstrap is:

> "since you do not have the whole population, do the best with what you do have, which is the observed sample".

In order to infer a population parameter, bootstrap technique uses the *plug-in* principle [20]. To understand it, suppose one wishes to estimate an important population parameter $\theta = \bar{\mathbf{E}}_i$. The population is unknown and only a random sample is available.

In order to find out θ, the plug-in principle uses the sample to calculate an estimator $\widehat{\theta}$, which corresponds to a function that produces an estimate for a population parameter. For example, if $\theta = \bar{\mathbf{E}}_i$, $\widehat{\theta} = \bar{\mathbf{E}}_i^*$. In other words, the sample estimator is used to infer the analogous unknown population parameter. Thus, based on the value of $\widehat{\theta}$, the parameter θ is inferred [20]. This is shown in Figure 6.1

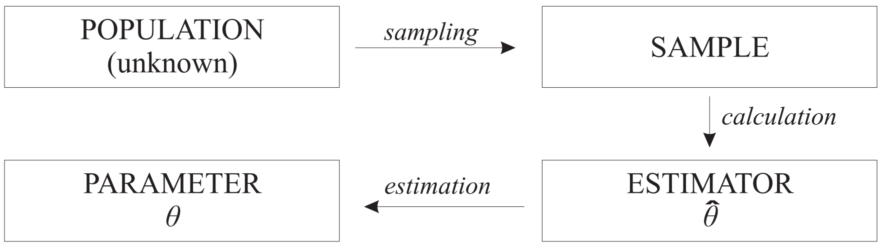


Figure 6.1. Illustration of the plug-in principle

Bootstrap is a direct application of the plug-in principle [20]. For this reason, it is capable of estimating several different population parameter.

As long as the estimator has been defined, its reliability can be assessed through bootstrap before it can be used to infer the population parameter. Indeed, such a technique aims at deriving robust estimates of standard errors of an estimator and confidence intervals of a population parameter. To do so, its strategy is based on resampling with replacement the original sample data.

Although there are other inference methods which are based on resampling, they may not be suitable for computing fault resilience values for a given system. For example, Monte Carlo simulation considers that the population parameter distribution is known, which may not be true when it comes to errors [46]. Jackknife technique, which is in fact easier to compute than bootstrap, could also be used. However, since this method uses a smaller number of replicas than bootstrap, it uses limited information about the estimator. For this reason, Jackknife is considered as an approximation of bootstrap [20].

Bootstrap is a completely automatic method, whose application is very simple and requires no theoretical calculations. Also, such a technique may provide more accurate confidence intervals for standard parameters such as the mean, if compared to normal approximation, used in traditional inference methods [46]. Moreover, since bootstrap is based on resampling the original sample data, it allows to infer population parameters even from small samples with unknown distribution patterns.

Although resampling is an important characteristic, since the generated samples represent a potential set of observations, it can be too CPU consuming depending on the sample size and on the number of necessary iterations. However, as computation power has been remarkably improving, this does not pose a major problem for current machines [12].

In order to understand the bootstrap algorithm, consider Figure 6.2. Let the estimator $\widehat{\theta}$ be the sample mean and assume that the available sample is defined as $\mathbf{x} = (x_1, x_2, \ldots, x_s)$, where s is the sample size. First, resampling is applied to generate B *bootstrap samples*, which are represented by $\mathbf{x}^{*i} = (x_1^*, x_2^*, \ldots, x_s^*)$, $i = 1, 2, \ldots, B$. Each tuple $\mathbf{x}^{*i}$ is a randomized version of the original sample $\mathbf{x}$, drawn with replacement. This means that any value x_j^* may appear several times in the sample $\mathbf{x}^{*i}$, and some may not appear at all. The number of bootstrap samples necessary to compute an estimator depends on the original sample size. Indeed, some authors assume that it usually varies from 25 to 300 samples [20], while others suggest 100 bootstrap samples to compute the standard error and 1000 bootstrap samples to compute the $(1-\alpha)\%$ confidence interval

[13].

Resampling is an important issue in bootstrap since the generated bootstrap samples represent a potential set of observations, which may be better than relying on standard assumptions, which may not be true.

The next step is then to calculate an estimator $\widehat{\theta^{*i}}$ for each bootstrap sample $\mathbf{x}^{*i}$. And finally, the sample estimated mean $\widehat{\theta^{*}}$ can be calculated, as the average mean of $\widehat{\theta^{*i}}$, as shown in Figure 6.2.

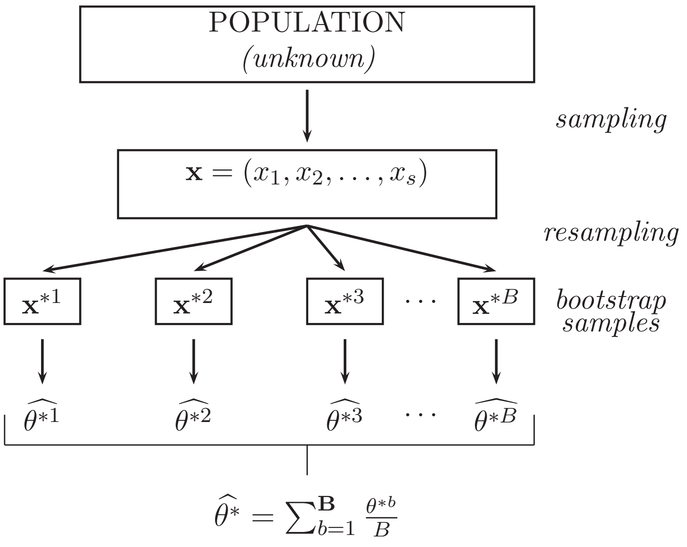


Figure 6.2. Illustration of bootstrap algorithm.

It is also important to compute the standard error of $\widehat{\theta^{*}}$ [20], which is given by

$$\widehat{se_{\mathbf{B}}} = \sqrt{\frac{\sum_{b=1}^{B}(\widehat{\theta^{*b}} - \widehat{\theta^{*}})^2}{B-1}}$$

The whole bootstrap procedure is shown in Algorithm 7.

Bootstrap algorithm was also applied to Example 6.1 considering both RM and EDF scheduling to estimate the mean effort $\bar{\mathbf{E}}_i$. Here, we decided to consider the samples generated previously (used to compute the mean $\bar{\mathbf{E}}_i$ in Section 6.1) as the original sample for comparison means. For such an example, we have set $B = 500$. The mean effort $\bar{\mathbf{E}}_i$

Algorithm 7: Bootstrap Algorithm

```
1 Select $B$ independent bootstrap samples;
2 Calculate the estimator for each generated bootstrap sample;
3 Compute the bootstrap estimator $\widehat{\theta}^{*}$;
4 Estimate the standard error $\widehat{se_B}$ from the sample standard deviation;
```

was computed using the **R** software [53], which implements bootstrap algorithm. The found results are shown in Table 6.3.

Table 6.3. $\bar{\mathbf{E}}_i$ with 95% of confidence according to bootstrap procedure

	RM				EDF			
i	$\bar{\mathbf{E}}_i^*$	$\widehat{se_B}$	CI_i	$\bar{\mathbf{E}}_i$	$\bar{\mathbf{E}}_i^*$	$\widehat{se_B}$	CI_i	$\bar{\mathbf{E}}_i$
1	1.000	0.0000	[1.000,1.000]	1.000	1.000	0.0000	[1.000,1.000]	0.999
2	0.614	0.0027	[0.608,0.619]	0.614	0.658	0.0028	[0.652,0.663]	0.661
3	0.432	0.0026	[0.427,0.437]	0.433	0.507	0.0030	[0.501,0.513]	0.507
4	0.337	0.0026	[0.332,0.342]	0.338	0.406	0.0030	[0.400,0.411]	0.407
5	0.294	0.0024	[0.289,0.298]	0.295	0.371	0.0031	[0.365,0.377]	0.375
6	0.262	0.0024	[0.257,0.267]	0.264	0.371	0.0031	[0.365,0.378]	0.373
7	0.247	0.0027	[0.242,0.252]	0.247	0.332	0.0023	[0.327,0.336]	0.332
8	0.206	0.0025	[0.201,0.211]	0.207	0.291	0.0027	[0.286,0.296]	0.291
9	0.173	0.0021	[0.168,0.177]	0.174	0.282	0.0035	[0.275,0.289]	0.280
10	0.162	0.0024	[0.157,0.167]	0.161	0.256	0.0035	[0.249,0.263]	0.256

The confidence interval CI_i for $\bar{\mathbf{E}}_i$ was computed considering a confidence of 95% and is also shown in Table 6.3. The standard error $\widehat{se_B}$ is also presented to evidence sample standard deviation. Both standard error and confidence bootstrap interval were automatically calculated using a bootstrap **R** function. Comparing the results from Tables 6.2 and 6.3, it is possible to observe that the main difference is presented for the confidence interval, which are closer to population parameter $\bar{\mathbf{E}}_i$ in Table 6.3, which is based on bootstrap. This is due to smaller standard errors obtaind from bootstrap.

Bootstrap can also be used to compute any other estimator, since its calculation procedure is the same for any of them. For example, it may be useful for designers to determine system resilience based on trust levels. Percentile estimator would be helpful in this situation since it represents the value bellow which a certain percent of observations fall. To illustrate this, we applied bootstrap algorithm to compute the $10th$ percentile for each task in Example 5.1. Such values are presented in Table 6.4.

Table 6.4. $10th$-percentile of $\mathbf{E}_i$ with 95% of confidence according to bootstrap procedure

	RM				EDF			
i	P^*_{10}	$\widehat{se_B}$	CI_i	P_{10}	P^*_{10}	$\widehat{se_B}$	CI_i	P_{10}
1	1.000	0.0000	[1.000,1.000]	1.000	1.000	0.0000	[1.000,1.000]	1.000
2	0.538	0.0000	[0.538,0.538]	0.538	0.583	0.2639	[0.539,0.615]	0.615
3	0.333	0.0138	[0.304,0.333]	0.333	0.375	0.0007	[0.370,0.380]	0.375
4	0.280	0.0048	[0.269,0.280]	0.280	0.321	0.0043	[0.321,0.333]	0.321
5	0.241	0.0047	[0.227,0.242]	0.241	0.303	0.0043	[0.290,0.313]	0.300
6	0.200	0.0059	[0.194,0.212]	0.206	0.282	0.0045	[0.276,0.290]	0.290
7	0.182	0.0027	[0.178,0.186]	0.182	0.272	0.0022	[0.273,0.279]	0.273
8	0.158	0.0044	[0.150,0.163]	0.158	0.224	0.0029	[0.220,0.228]	0.224
9	0.141	0.0030	[0.138,0.149]	0.138	0.233	0.0077	[0.222,0.246]	0.232
10	0.132	0.0040	[0.121,0.139]	0.128	0.218	0.0045	[0.207,0.222]	0.211

Although we assumed $10th$-percentile, any other estimator could be used. It is important to mention that the confidence interval computed for such an estimator was the percentile confidence interval [20]. Also, in this case, the $10th$-percentile for the effort regarding all simulation scenarios, denoted by P_{10}, was known and is also presented in Table 6.4.

In the following section simulation experiments are detailed and their results are discussed.

6.3 RESULTS DISCUSSION

In order to illustrate how the derived fault resilience metric can be used to compare different real-time systems, we present some obtained simulation results. Section 6.3.1 presents the characteristics of the simulated task sets and makes a detailed comparison between RM and EDF scheduled systems, based on the mean effort $\bar{\mathbf{E}}^*_i$ of each task τ_i. The confidence interval for $\bar{\mathbf{E}}_i$ is determined considering both traditional inference and bootstrap technique in order to compare the results of such methods. Bootstrap technique is also used to compute the confidence interval for the $25th$, $50th$ and $75th$ percentiles. Such results are presented in Section 6.3.2

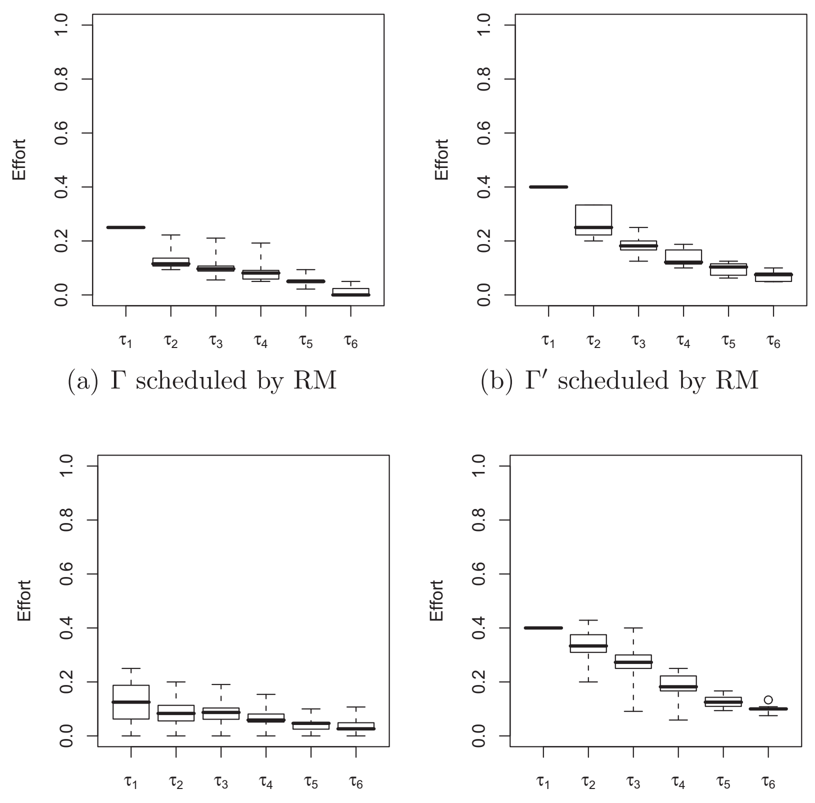
6.3.1 RM *vs.* EDF From Fault Resilience Viewpoint

We have generated 60 task sets composed of six tasks, each. Task periods ($\mathbf{T}$) and execution times ($\mathbf{C}$) were randomly generated so that their processor utilization factor were bounded in predefined intervals. To do so, each task period T_i and execution cost C_i were generated so that $T_i \in [1; 50]$ and $C_i \in [1; 5]$. The intervals defined for the processor utilization factor were $[50; 60)$, $[60; 70)$, $[70; 80)$ and $[80; 90)$. Also, each of these intervals contains 15 task sets. Tasks in each task set are ordered according to their periods. Thus, $T_1 \leq T_2 \leq \ldots \leq T_6$.

Initially, we assumed the mean effort $\bar{\mathbf{E}}_i$ as the fault resilience metric. Since the population of simulation scenarios Ω_i is unknown, we computed the sample size $|\Omega_i^*|$ for each task $\tau_i \in \Gamma$. We set $\alpha = 5\%$, the sample error $|\bar{\mathbf{E}}_i^* - \bar{\mathbf{E}}_i| = 5\text{x}10^{-3}$ and the standard deviation σ_i for each $\tau_i \in \Gamma$ was assumed to be known. Each sample of simulation scenarios was given as input to the simulation engine and results are discussed throughout this section.

In order to compare RM and EDF from fault resilience viewpoint, first we considered two different task sets Γ and Γ', generated as described earlier, with utilization factor bounded within $[80; 90)$. Figure 6.3 shows the empirical distribution of the effort through *boxplot* diagrams, which indicate the first, second and third quartiles of a distribution as well as their minimum and maximum values.

Considering the second quartile, which is equivalent to the median and is represented by bold lines in Figure 6.3, most effort values for RM obtained for Γ' are greater than the ones obtained for Γ. This can be seen in Figures 6.3(a) and 6.3(b). Such a result indicates that Γ' is more resilient to faults than Γ, regarding RM. Both Γ and Γ' were also submitted to EDF an the result is shown in Figures 6.3(c) and 6.3(d). Considering



(a) Γ scheduled by RM

(b) Γ′ scheduled by RM

(c) Γ scheduled by EDF

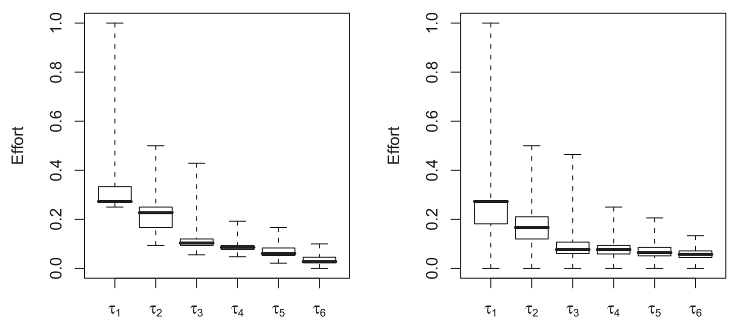
(d) Γ′ scheduled by EDF

Figure 6.3. Fault resilience distribution for Γ and Γ′

EDF, it is possible to observe that Γ′ was also more resilient to faults than Γ.

In order to compare RM and EDF, we considered simulating 15 task sets with processor utilization factor bounded within $[80; 90)$. The empirical distribution of the effort is presented in Figure 6.4. Observe that considering the median, the fault resilience for τ_1, τ_2, τ_3 and τ_4 are greater in RM than in EDF. Also, the difference between the minimum and maximum values are greater in EDF than in RM, which denotes that variability of effort values is greater in EDF. Further, notice that regarding task τ_5, the effort values are very close in both RM and EDF and for τ_6, EDF was more resilient to faults than RM. Such a result can be numerically observed in Table 6.5, which presents the mean effort $\bar{\mathbf{E}}_i^*$ for all task sets with processor utilization within $[80; 90)$, regarding both RM and EDF.

We decided to analyze the mean effort $\bar{\mathbf{E}}_i^*$ of each task τ_i regarding the processor



(a) RM (b) EDF

Figure 6.4. Fault resilience distribution for task sets with utilization $\mathbf{U} \in (80; 90]$

Table 6.5. Mean effort for RM and EDF

	$\bar{\mathbf{E}}_i$					
i	1	2	3	4	5	6
RM	0.413	0.226	0.120	0.087	0.069	0.031
EDF	0.348	0.175	0.091	0.077	0.069	0.059

utilization factor. To do so, we (a) selected an interval for the processor utilization factor; (b) simulated all task sets whose processor utilization factor were within the chosen interval and (c) computed the mean effort $\bar{\mathbf{E}}_i$ for each task τ_i.

Figure 6.5 shows the mean effort $\bar{\mathbf{E}}_1^*$ of τ_1 considering both RM and EDF and systems with different processor utilization levels. Observe that considering task τ_1, RM presented a better fault resilience than EDF for most processor utilization ranges. This was an expected result, since for RM higher priority tasks always preempts lower priority ones, while in EDF this does not necessarily happens. Also, notice that for task sets with utilization bounded within $[50; 60)$ RM and EDF presented almost the same fault resilience.

The intuition behind such a comparison is that the greater the processor utilization the smaller the processor idle time, and consequently the smaller the effort $\mathbf{E}_i$. Now, observe Figure 6.6, which shows the mean effort $\bar{\mathbf{E}}_{\mathbf{2}}^*$ related to task τ_2 considering different processor utilization ranges. Note that regarding the interval $[70; 90)$ RM was more resilient to faults than EDF, while for task sets with utilization within $[50; 70)$ EDF was

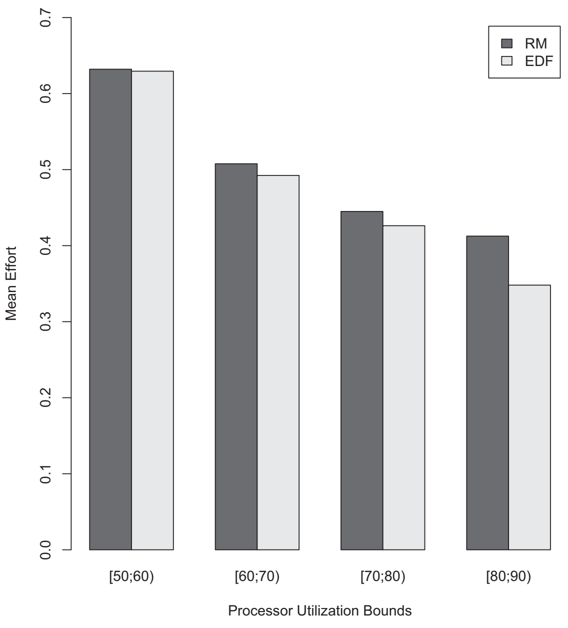


Figure 6.5. Mean effort $\mathbf{\bar{E}_1^*}$ for τ_1 regarding different processor utilization levels

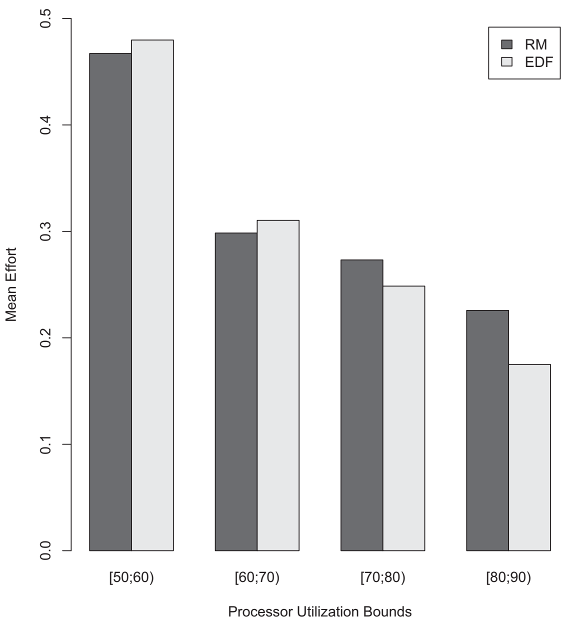


Figure 6.6. Mean effort $\mathbf{\bar{E}_2^*}$ for τ_2 regarding different processor utilization levels

more resilient to faults than RM. Until now, RM was more resilient to faults than EDF. However, since the priority of the analyzed job is beginning to become lower, such a result must change. Indeed, if we compute the fault resilience for τ_2, without considering the utilization ranges, we can note that the mean effort $\bar{\mathbf{E}}^*_{\mathbf{2}} = 0.42$ for RM and $\bar{\mathbf{E}}^*_{\mathbf{2}} = 0.43$ for EDF, which evidences that for τ_2 EDF is slightly more resilient to fault than RM.

Figure 6.7 shows the mean effort for task τ_3 considering both RM and EDF. Considering

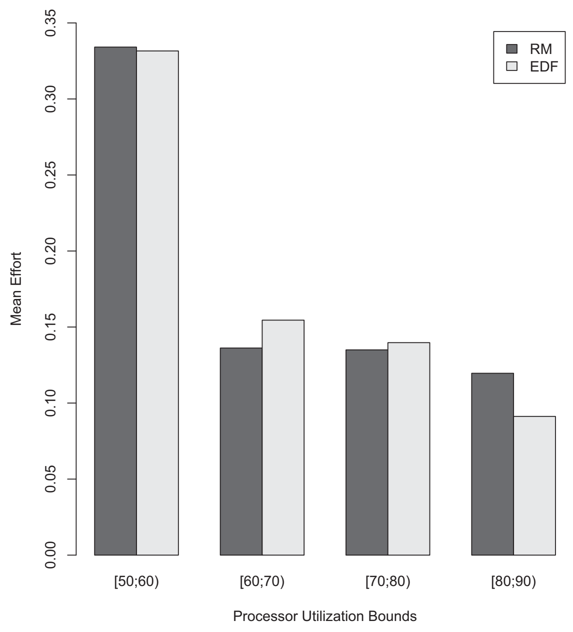


Figure 6.7. Mean effort $\bar{\mathbf{E}}^*_{\mathbf{3}}$ for τ_3 regarding different processor utilization levels

task sets with utilization within $[80; 90)$, the mean effort was greater in RM than in EDF. However, this was not observed for task sets with utilization within $[60; 80)$, where EDF presented a better overall behavior. Also, notice that when the processor utilization is bounded within $[50; 60)$ RM was a little more resilient to faults than EDF. If we compute the fault resilience for τ_3, for all ranges, we can note that the mean effort $\bar{\mathbf{E}}^*_3 = 0.22$ for RM and $\bar{\mathbf{E}}^*_3 = 0.23$ for EDF, regarding the considered task sets.

For now, let us observe what happens when the analyzed job is related to τ_4. This is shown in Figure 6.8, which presents $\bar{\mathbf{E}}_4$ for RM and EDF. Observe that in this case, EDF presents a better overall behavior for all simulated systems, whose processor utilization is in $[50; 80)$. The exception was for task sets in $[80; 90)$, in which RM was more resilient to faults. In spite of that, the fault resilience for τ_4, without considering the utilization bounds for RM is $\mathbf{\bar{E}_4^*} = 0.13$, while for EDF $\mathbf{\bar{E}_4^*} = 0.15$.

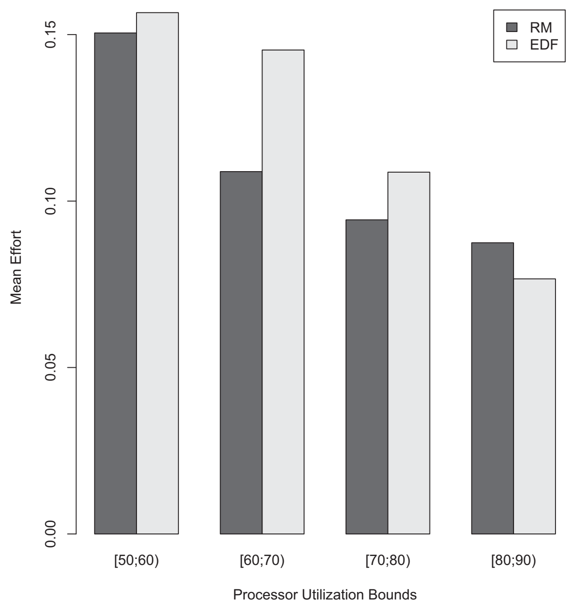


Figure 6.8. Mean effort $\mathbf{\bar{E}_4^*}$ for τ_4 regarding different processor utilization levels

Figure 6.9 presents the mean effort for task τ_5 for RM and EDF scheduled systems.

Notice that for Figure 6.9, the mean effort $\mathbf{\bar{E}_5^*}$ is greater in EDF than in RM. Also, the difference between EDF and RM mean effort is more evident for all task sets with utilization between $[50; 80)$. Actually, the mean effort for task sets with utilization bounded within $[80; 90)$ is almost the same. As mentioned before, as the priority of jobs become lower, EDF scheduled task sets become more resilient to faults than those scheduled by RM. Indeed, the fault resilience for τ_5, without considering the utilization bounds for RM is $\mathbf{\bar{E}_5^*} = 0.10$, while for EDF $\mathbf{\bar{E}_5^*} = 0.14$.

Last but not least, let us consider the mean effort for τ_6, which is presented in Figure 6.10. Observe that in this case, since the jobs of τ_6 are the ones with lower priorities, EDF is more resilient than RM for all considered task sets. Also, the difference between the mean effort values are more evident. Actually, $\mathbf{\bar{E}_6^*} = 0.08$ for RM, while for EDF

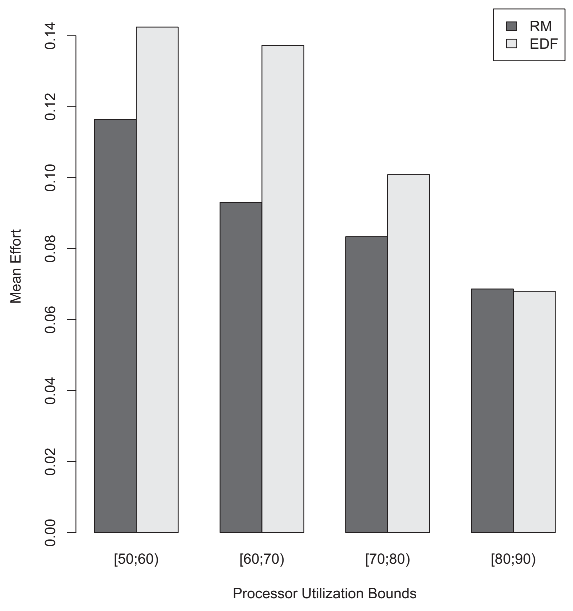


Figure 6.9. Mean effort $\bar{\mathbf{E}}_{\mathbf{5}}^{*}$ for τ_5 regarding different processor utilization levels

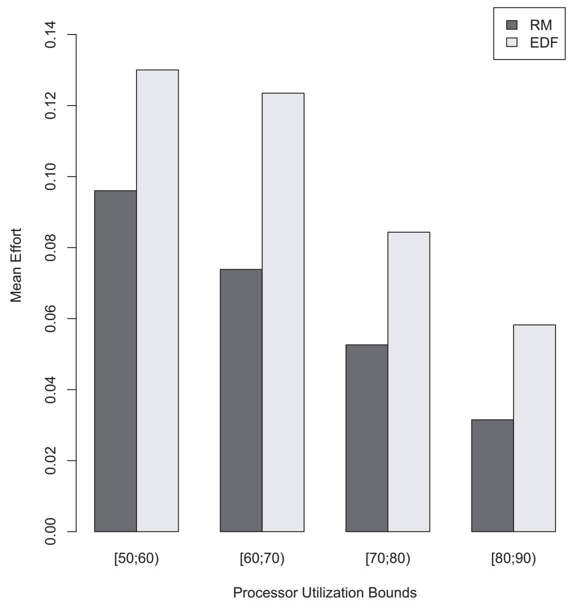


Figure 6.10. Mean effort $\bar{\mathbf{E}}_{\mathbf{6}}^{*}$ for τ_6 regarding different processor utilization levels

$\mathbf{\bar{E}}_\mathbf{6}^* = 0.12$. A comparison between all generated systems, considering both RM and EDF is shown in Figure 6.11.

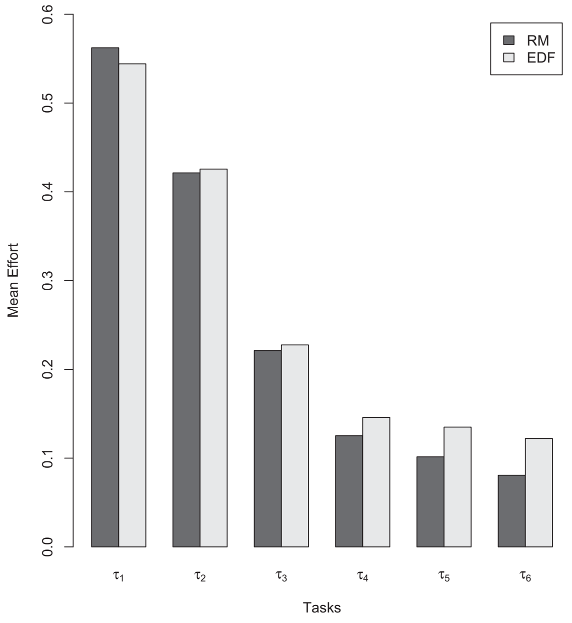


Figure 6.11. Mean effort $\mathbf{\bar{E}}_i^*$ for RM and EDF

Notice that, as expected, the mean effort $\mathbf{\bar{E}}_i^*$ is greater for tasks with highest priorities according to RM, since no other task interfere in its execution, and consequently more errors have to be generated. On the other hand, lower priority tasks are preempted by any higher priority task and so the effort $\mathbf{\bar{E}}_i^*$ must be smaller. Also, observe that RM tends to be better than EDF only for lower priority tasks, as explained earlier. For most tasks, the fault resilience for EDF gets better than for RM, in general case.

The following section shows the confidence interval for the mean effort according to both traditional and bootstrap inference methods.

6.3.2 Computing Confidence Intervals

In this section we present the confidence intervals for the mean effort and percentile parameters, considering the simulated task sets. First, we show the confidence intervals for the mean effort $\bar{\mathbf{E}}_i^*$, regarding all processor utilization ranges, computed according to traditional inference, using **R** software. This is presented in Table 6.6

Table 6.6. $\bar{\mathbf{E}}_\mathbf{i}^*$ with 95% of confidence according to traditional inference

	RM		EDF	
i	$\bar{\mathbf{E}}_\mathbf{i}^*$	CI_i	$\bar{\mathbf{E}}_\mathbf{i}^*$	CI_i
1	0.562	[0.549,0.558]	0.544	[0.535,0.545]
2	0.421	[0.420,0.423]	0.426	[0.424,0.427]
3	0.221	[0.220,0.222]	0.228	[0.227,0.228]
4	0.125	[0.125,0.125]	0.149	[0.146,0.146]
5	0.101	[0.101,0.102]	0.135	[0.135,0.135]
6	0.080	[0.080,0.081]	0.122	[0.122,0.122]

The confidence interval was computed considering a confidence of 95%. In Table 6.6 it is possible to observe numerically that EDF is more resilient than RM. Indeed, the mean effort values $\bar{\mathbf{E}}_i^*$ presented in the table are the same discussed in the previous section. Also, as mentioned before, RM is more resilient than EDF only for task τ_1 because of its priority.

The confidence intervals for the mean effort were also computed considering bootstrap with a confidence of 95%. Such results are presented in Table 6.7.

Table 6.7. $\bar{\mathbf{E}}_\mathbf{i}^*$ with 95% of confidence according to bootstrap procedure

	RM			EDF		
i	$\bar{\mathbf{E}}_\mathbf{i}^*$	CI_i	$\widehat{se_\mathbf{B}}$	$\bar{\mathbf{E}}_\mathbf{i}^*$	CI_i	$\widehat{se_\mathbf{B}}$
1	0.562	[0.566,0.568]	0.0024	0.544	[0.539,0.549]	0.0023
2	0.309	[0.309,0.311]	0.0005	0.312	[0.311,0.313]	0.0005
3	0.221	[0.220,0.222]	0.0004	0.228	[0.227,0.228]	0.0004
4	0.125	[0.125,0.125]	0.0001	0.146	[0.146,0.146]	0.0001
5	0.101	[0.101,0.101]	0.0001	0.135	[0.135,0.135]	0.0001
6	0.081	[0.081,0.081]	0.0001	0.122	[0.122,0.122]	0.0001

Observe that, regarding task τ_1, the mean effort for RM is greater than that for EDF, which is not true for the other tasks. Also, as mentioned before, the confidence interval for $\bar{\mathbf{E}}_i^*$ computed by bootstrap is tighter than the ones computed by traditional inference, since bootstrap provides more accurate standard errors for the estimators, such as the sample mean.

Bootstrap was also used to compute the percentile confidence interval for percentile estimators, considering both RM and EDF. Table 6.8 shows the result for each task τ_i considering 25th-percentile. Results presented in column $\mathbf{P}^*_{25}$ indicates the estimated value, bellow which 25% of the effort values appear.

Table 6.8. 25th-percentile with 95% of confidence according to bootstrap procedure

	RM			EDF		
i	$\mathbf{P}^*_{25}$	CI_i	$\widehat{se_{\mathbf{B}}}$	$\mathbf{P}^*_{25}$	CI_i	$\widehat{se_{\mathbf{B}}}$
1	0.313	-	0	0.313	-	0
2	0.156	-	0	0.147	-	0
3	0.122	[0.122,0.122]	0.0001	0.136	-	0
4	0.095	-	0	0.116	-	0
5	0.085	-	0	0.111	-	0
6	0.061	-	0	0.100	-	0

Observe that the 25th-percentile values are greater in EDF, except for tasks τ_1, in which such values are the same, and τ_2 in which RM is more resilient than EDF. Also, since $\mathbf{P}^*_{25}$ values presented no variability, the standard error was zero for most tasks and consequently, there was no confidence interval for τ_1, τ_2, τ_4, τ_5 and τ_6.

The 50th-percentile were also computed and is presented in Table 6.9. Notice that for task τ_2 RM was also more resilient to faults than EDF. Considering tasks τ_1, τ_2 and τ_3 the standard error was zero and no confidence interval could be calculated, which was not true for τ_5 and τ_6.

Finally, we computed the 75th-percentile, which is presented in Table 6.10. Observe that in this case all values for EDF were greater than RM values. Also, the standard error for tasks τ_4 and τ_5 is zero and consequently there is no confidence interval.

Table 6.9. $50th$-percentile with 95% of confidence according to bootstrap procedure

	RM			EDF		
i	$\mathbf{P}^*_{50}$	CI_i	$\widehat{se_{\mathbf{B}}}$	$\mathbf{P}^*_{50}$	CI_i	$\widehat{se_{\mathbf{B}}}$
1	0.500	-	0	0.500	-	0
2	0.217	-	0	0.192	-	0
3	0.152	-	0	0.166	-	0
4	0.114	[0.114,0.114]	0.0001	0.143	-	0
5	0.095	[0.095,0.095]	0.0001	0.136	[0.135,0.136]	0.0004
6	0.077	[0.076,0.076]	0.0001	0.122	[0.122,0.122]	0.0001

Table 6.10. $75th$-percentile with 95% of confidence according to bootstrap procedure

	RM			EDF		
i	$\mathbf{P}^*_{75}$	CI_i	$\widehat{se_{\mathbf{B}}}$	$\mathbf{P}^*_{75}$	CI_i	$\widehat{se_{\mathbf{B}}}$
1	1.000	[0.500,1.000]	0.1617	0.538	[0.500,0.846]	0.1071
2	0.353	[0.353,0.357]	0.0065	0.375	[0.368,0.375]	0.0030
3	0.219	[0.214,0.219]	0.0010	0.231	[0.230,0.233]	0.0012
4	0.143	-	0	0.167	-	0
5	0.118	-	0	0.158	-	0
6	0.098	[0.096,0.098]	0.0003	0.143	-	0

6.4 SUMMARY

This chapter presented two statistical inference methods, which were used to infer the fault resilience of a given task based on a sample of simulation scenarios. Traditional inference was used to compute the confidence interval for the mean effort, while bootstrap technique computed the confidence interval for both mean effort and percentile estimators.

Results presented in this section show how to compare two classical scheduling policies, RM and EDF, using a fault resilience metric. It was possible to compute fault resilience from each task view point, which can be very useful, considering that each task may have a different criticality. It is important to mention that the approach described in this work can deal with other scheduling policies. To do so, it is necessary to modify the simulation engine accordingly.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this work we derived a metric, called error generator *effort*, which aims at measuring the fault resilience of real-time systems. Such a metric is designed to be as independent of the assumed system model and/or fault model as possible. Moreover, it can be used to subsidize the system designer decisions when choosing the fault-tolerant mechanisms that best suit their systems.

In order to compute the fault resilience according to the proposed metric, a simulation environment was built. The developed framework has the advantage of not needing to simulate the system during the whole hyperperiod, which may be too time consuming in general case. The basic idea is to simulate the system during specific time windows and then compute the effort for each of them.

The simulation time windows are derived based on the concept of simulation scenarios, which are generated by a framework component called *scenario generator*. Two kinds of procedures were implemented to generate simulation scenarios: (a) sequential algorithms, which outputs all possible simulation scenarios for a given task and (b) random procedures, which generates a random subset of simulation scenarios.

After simulation scenarios are generated they are given as input to the *simulation engine*, which is responsible for computing the effort for a given system task. It is important to mention that even considering specific time windows, there may be some situations in which deriving the fault resilience for all simulation scenarios may not be practical. In this case, statistical inference methods are used so that the effort for a given task can be inferred based on a sample of such values.

In this work we considered two statistical inference methods. First, *traditional inference*, which is based on the Central Limit Theorem, was used to compute the confidence intervals for the mean effort, regarding each system task. *Bootstrap*, which is a computational statistical inference method, was considered for determining the confidence interval for the mean effort and also for other parameters.

Experimental results indicate the differences of two well known scheduling policies, RM and EDF, from the viewpoint of fault resilience, demonstrating that our approach is reasonably independent of the system model.

Future work includes evaluating extensions of the proposed analysis focusing on two main aspects. First, better strategies for decreasing the pessimism of the analysis should be investigated. Less restrictive task models and the incorporation of probabilistic behavior for the error generator should also be addressed. A more ambitious goal is to use the principles described here to derive probabilistic schedulability bounds for real-time systems.

REFERENCES

[1] A. Armoush, F. Salewski, and S. Kowalewski. A Hybrid Fault Tolerance Method for Recovery Block with a Weak Acceptance Test. In *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC '08)*, volume 1, pages 484 – 491, December 2008.

[2] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, 8:284 – 292, 1993.

[3] Algirdas Avizienis, Jean-Claude Laprie, Carl Landwehr, and Brian Randell. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[4] Hakan Aydin. Exact Fault-Sensitive Feasibility Analysis of Real-Time Tasks. *IEEE Transactions on Computers*, 56(10):1372 – 1386, 2007.

[5] Hakan Aydin, Rami Melhem, and Daniel Mossé. Tolerating Faults while Maximizing Reward. *Proceedings of the Twelfth Euromicro Conference on Real-Time Systems (Euromicro'00)*, pages 219 – 226, June 2000.

[6] Alan Burns. Scheduling Hard Real-Time Systems: A Review. *Software Engineering Journal*, 6(3):116–128, 1991.

[7] Alan Burns, Guillem Bernat, and Ian Broster. A Probabilistic Framework for Schedulability Analysis. *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT' 03*, pages 1 – 15, 2003.

[8] Alan Burns, Rob Davis, and Sasikumar Punnekkat. Feasibility Analysis of Fault Tolerant Real-Time Task Sets. In *Euromicro Real-Time Systems Workshop*, pages 29–33, L'Aquila, Italy, June 1996. IEEE Society Press.

[9] Alan Burns, Sasikumar Punnekkat, L. Strigini, and D. R. Wright. Probabilistic Scheduling Guarantees for Fault-Tolerant Real-Time Systems. In *Proceedings of the conference on Dependable Computing for Critical Applications (DCCA '99)*, page 361, Washington, DC, USA, 1999. IEEE Computer Society.

[10] Giorgio C. Buttazzo. Rate Monotonic vs. EDF: Judgment Day. *Real-Time Systems*, 29(1):5–26, 2005.

[11] Marco Caccamo and Giorgio Buttazzo. Optimal Scheduling for Fault-Tolerant and Firm Real-Time Systems. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications(RTCSA '98)*, page 223, Washington, DC, USA, 1998. IEEE Computer Society.

[12] Russell C. H. Cheng. Bootstrap Methods in Computer Simulation Experiments. In *WSC '95: Proceedings of the 27th conference on Winter simulation*, pages 171–177, Washington, DC, USA, 1995. IEEE Computer Society.

[13] Michael R. Chernick. *Bootstrap Methods: A Guide for Practitioners and Researchers.* Wiley, second edition, 2008.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* Elsevier, second edition, 2002.

[15] José Luis Díaz, José María López, and Daniel Fernando García. Probabilistic Analysis of the Response Time in a Real-Time System. In *1st Workshop on Advanced Real-Time Technologies*, Aranjuez, Spain, 2002.

[16] José Luis Díaz, José María López, and Daniel Fernando García. Stochastic Analysis of the Steady-State Backlog in Periodical Real-Time Systems. Technical Report TR-03-SASS, Departamento de Informática, University of Oviedo, 2003.

[17] George Marconi de Araújo Lima and Alan Burns. An Optimal Fixed-Priority Assignment Algorithm for Supporting Fault-Tolerant Hard Real-Time Systems. *IEEE Transactions on Computers*, 52(10):1332–1346, 2003.

[18] Edinaldo O. de Jesus and George Marconi de Araújo Lima. Escalonamento para Sistemas de Tempo Real Tolerantes a Falhas: Um Estudo Empírico. *Proceedings of the 7th Brazilian Workshop on Real-Time and Embedded Systems (WTR'05)*, 2005.

[19] Rômulo Silva de Oliveira and Joni da Silva Fraga. Uma Solução Mista para o Escalonamento Baseado em Prioridades de Aplicações de Tempo Real Críticas. *Congresso da Sociedade Brasileira de Computação (SEMISH'96)*, 1996.

[20] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap.* Chapman & Hall / CRC, 1993.

[21] Joaquin Entrialgo, Javier Garcia, Jose Luis Diaz, and Daniel Fernando Garcia. Stochastic Metrics for Debugging the Timing Behaviour of Real-Time Systems. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium(RTAS '07)*, pages 183–192, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Jean-Marie Farines, Joni da Silva Fraga, and Rômulo Silva de Oliveira. *Sistemas de Tempo Real.* Departamento de Automação e Sistemas - Universidade Federal de Santa Catarina, Florianópolis, Santa Catarina, 2000.

[23] Sunondo Ghosh, Rami Melhem, and Daniel Mossé. Enhancing Real-Time Schedules to Tolerate Transient Faults. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, page 120, Washington, DC, USA, 1995. IEEE Computer Society.

[24] Sunondo Ghosh, Rami G. Melhem, Daniel Mossé, and Joydeep Sen Sarma. Fault-Tolerant Rate-Monotonic Scheduling. *Real-Time Systems*, 15(2):149–181, 1998.

[25] Ching-Chih Han, Kang G. Shin, and Jian Wu. A Fault-Tolerant Scheduling Algorithm for Real-Time Periodic Tasks with Possible Software Faults. *IEEE Transactions on Computers*, 52(3):362–372, 2003.

[26] Joel Huselius, Johan Kraft, Hans Hansson, and Sasikumar Punnekkat. Evaluating the Quality of Models Extracted from Embedded Real-Time Software. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS '07)*, pages 577–585, Washington, DC, USA, 2007. IEEE Computer Society.

[27] Rolf Isermann. Modeling and Design Methodology for Mechatronic Systems. *IEEE/ASME Transactions on Mechatronics*, 1(1):16 – 28, March 1996.

[28] Li Jun, Yang Fumin, and Lu Yansheng. A Feasible Schedulability Analysis for Fault-Tolerant Hard Real-Time Systems. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 176–183, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Kim Kanghee, José Luis Díaz, L. L Bello, José María López, Chang-Gun Lee, and Sang Lyul Min. An Exact Stochastic Analysis of Priority-Driven Periodic Real-Time Systems and its Approximations. *Transactions on Computers*, 54(11):1460–1466, 2005.

[30] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1981.

[31] John Lehoczky, Liu Sha, and Ye Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of Real Time Systems Symposium*, pages 166–171, Santa Monica, California, USA, 1989.

[32] Frank Liberato, Rami Melhem, and Daniel Mossé. Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems. *IEEE Transactions on Computers*, 49(9):906–914, 2000.

[33] George Lima and Alan Burns. Scheduling Fixed-Priority Hard Real-Time Tasks in the Presence of Faults. In *Proceedings of the 2nd Latin-American Symposium on Dependable Computing (LNCS'05)*, volume 3747, pages 154–173. Springer-Verlag, 2005.

[34] George Lima and Flávia Maristela S. Nascimento. Simulation Scenarios: a Means of Deriving Fault Resilience for Real-Time Systems. *Proceedings of the 11th Brazilian Workshop on Real-Time and Embedded Systems (WTR'09)*, 2009.

[35] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[36] Ren C. Luo. Sensors and Actuators for Intelligent Mechatronic Systems. In *Proceedings of the 27th Annual Conference of the IEEE Industrial Electronics Society (IECON '01)*, volume 3, pages 2062 – 2065, 2001.

[37] Raimundo Macêdo, George Lima, Luciano Barreto, Aline Andrade, Alírio Sá, Frederico Barboza, Rodrigo Albuquerque, and Sandro Andrade. Tratando a Previsibilidade em Sistemas de Tempo-Real Distribuídos: Especificação, Linguagens, Middleware e Mecanismos Básicos. In *22nd Brazilian Symposium on Computers Network and Distributed Systems (SBRC' 2004)*, pages 105–163. SBRC, 2004.

[38] Pedro Mejía-Alvarez, Hakan Aydin, Daniel Mossé, and Rami Melhem. Scheduling Optional Computations in Fault-Tolerant Real-Time Systems. In *Proceedings of the Seventh International Conference on Real-Time Systems and Applications (RTCSA'00)*, pages 323 – 330, Washington, DC, USA, 2000. IEEE Computer Society.

[39] Paulo Eigi Miyagi and Emilia Villani. Mecatrônica como Solução de Automação. *Revista Ciências Exatas*, 9/10(1-2):53 – 59, 2004.

[40] Pedro Morettin and Wilton Bussab. *Estatística Básica*. Saraiva, 2004.

[41] Daniel Mossé, Rami Melhem, and Sunondo Ghosh. A Nonpreemptive Real-Time Scheduler with Recovery from Transient Faults and Its Implementation. *IEEE Transactions on Software Engineering*, 29(8):752–767, 2003.

[42] Flávia Maristela S. Nascimento, George Lima, and Verônica Cadena Lima. Simulation-Based Analysis to Derive Fault Resilience in Real-Time Systems. *Proceedings of the Work in Progress Section of the 10th Brazilian Workshop on Real-Time and Embedded Systems (WTR'08)*, 2008.

[43] Flávia Maristela S. Nascimento, George Lima, and Verônica Cadena Lima. Deriving a Fault Resilience Metric for Real-Time Systems. *Proceedings of the 10th Brazilian Workshop on Tests and Fault Tolerance (WTF'09)*, 2009.

[44] Mihir Pandya and Miroslaw Malek. Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks. *IEEE Transactions on Computers*, 47(10):1102 – 1112, 1998.

[45] Nuno Pereira, Eduardo Tovar, Berta Batista, Luis Miguel Pinho, and Ian Broster. A Few What-Ifs on Using Statistical Analysis of Stochastic Simulation Runs to Extract Timeliness Properties. In *1st International Workshop on Probabilistic Analysis Techniques for Real-time and Embedded Systems (PARTES '2004)*, Pisa, Italy, 2004.

[46] Dimitris N. Politis. Computer-intensive Methods in Statistical Analysis. *IEEE Signal Processing Magazine*, 15:39–55, 1998.

[47] Krithi Ramamritham and John A. Stankovic. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. *Proceedings of the IEEE*, 82(1):55–67, 1994.

[48] Lui Sha, Tarek Abdelzaher, Karl-Erik Arzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[49] Larry Sieh, Peter Haniak, and Paul Richardson. Implementing Transient Fault Tolerance in Embedded Real-Time Systems, 2001.

[50] John A. Stankovic. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *Computer*, 21(10):10–19, 1988.

[51] John A. Stankovic, Chenyang Lu, and Sang H. Son. The Case for Feedback Control Real-Time Scheduling. *Proceedings of the 11th Euromicro Conference on Real-Time Systems (Euromicro'99)*, pages 11 – 20, 1998.

[52] John A. Stankovic and Fuxing Wang. The Integration of Scheduling and Fault Tolerance in Real-Time Systems. Technical report, University of Massachusetts, Amherst, MA, USA, 1992.

[53] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2009. ISBN 3-900051-07-0.

[54] Mario F. Triola. *Elementary Statistics.* Pearson, 2008.

[55] Anders Wall, Johan Andersson, and Christer Norstrom. Probabilistic Simulation-Based Analysis of Complex Real-Time Systems. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, pages 257–266, Washington, DC, USA, 2003. IEEE Computer Society.

[56] Taisy Silva Weber. *Tolerância a Falhas: Conceitos e Exemplos.* Instituto de Informática - Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul, 2001.