Design Optimisation of Fault-Tolerant Event-Triggered Embedded Systems

Jarik Poplavski Kany
and
Sigurd Hilbert Madsen

Supervisor: Paul Pop

Kongens Lyngby 2007
IMM-M.Sc.-2007-72

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

Computers today are getting smaller and cheaper and are almost everywhere in our daily lives: at our homes, in the cars, airplanes and industry – almost all devices we use contains one or more embedded computers. With growing usage of embedded devices the requirements are getting tighter. In this thesis we address safety-critical embedded systems, where not only the correct result, but also satisfying timing requirements of the system is vital even in the presence of faults.

The increase in computational speed and circuit density has raised the probability of having transient faults. Embedded systems that are used in safety-critical applications must be able to tolerate the increasing number of transient faults. If not, they might lead to failures that would have disastrous consequences and potentially endanger human lives and the environment.

This thesis addresses design optimisation for fault-tolerant event-triggered embedded systems. The hardware of these systems consists of distributed processing elements connected with communication busses. The applications to be run on the hardware are represented by directed acyclic graphs. Processes are scheduled using a fixed-priority preemptive scheduling policy, while messages are transmitted using the Controller Area Network bus protocol. Faults are tolerated for each process through either reexecution or active replication.

In this thesis we describe a model for representing fault-tolerant applications, called fault-tolerant process graphs (FTPG). We first propose schedulability analysis techniques which can determine whether a fault-tolerant application represented using an FTPG is schedulable. Three different approaches to the schedulability analysis have been proposed: ignoring conditions (IC), condition separation (CS) and brute force analysis (BF). They differ in the quality of the results and their runtime. Considering the response-time analysis, we also present an optimisation heuristic that decides for each process which fault-tolerance policy to use, and on which processing element to execute it, such that the application is schedulable.

We have evaluated the proposed schedulability analysis and optimisation methods using randomly-generated synthetic applications and a cruise controller application from the automotive industry.

Resumé

I dag er computere blevet så hurtige, små og billige, at vi nu er begyndt at bruge dem næsten alle steder i vores dagligdag: i hjemmet, i biler, i flyvemaskiner og på fabrikker – stort set alt elektronik indeholder en eller flere indlejrede computere. I takt med den stigende brug af indlejrede systemer vokser også kravet til deres pålidelighed. I denne afhandling adresserer vi sikkerhedskritiske indlejrede systemer, hvor ikke kun det rigtige resultat, men også overholdelse af tidsfristerne, er meget vigtigt, selv når der sker fejl.

Voksende klokfrekvenser og densiteten af digitale kredsløb har medført en øget sandsynlighed for transiente fejl. Indlejrede systemer, som bliver brugt til sikkerhedskritiske opgaver, skal være i stand til at modstå det stigende antal transiente fejl. Alternativt kan det medføre fatale konsekvenser, hvor der vil være fare for tab af menneskeliv eller miljøforurening.

Denne afhandling omhandler designoptimering af sikkerhedskritiske hændelsesstyrede indlejrede systemer. Hardwaren i disse systemer består af distribuerede enheder, som kommunikerer over kommunikationsbusser. Softwaren, som skal afvikles på den givne hardware, er repræsenteret som orienterede acykliske grafer. Processer bliver scheduleret ved brug af faste prioriteter og kan blive preempted af andre processer i applikationen. Beskeder bliver overført ved hjælp af Control Area Network protokol. Fejlene bliver tolereret for hver proces ved hjælp af enten reeksekvering eller replikering.

Vi beskriver en model til at præsentere fejltolerante applikationer – fejltolerante procesgrafer (FTPG). Vi foreslår en responstidsanlyse, som kan afgøre, om en fejltolerant applikation er schedulerbar. Tre forskellige tilgange bliver præsenteret: at ignorere fejlbetingelser (IC), med separering af fejlbetingelser

(CS) og den såkaldte “brute force” analyse (BF). Disse tilgange producerer resultater, som er forskellige både i kvaliteten og i den tid, der er nødvendig for at beregne dem. Baseret på responstidsanalysen, præsenterer vi også en optimerings-heuristik, der for hver process skal finde den optimale fejltolerance-teknik og afgøre på hvilken enhed, processen skal afvikles. Det skal sikre, at applikationen er schedulerbar.

Vi har evalueret de foreslåede responstidsanalyser og optimeringsheuristikken med tilfældigt genererede syntetiske applikationer og en fartpilot-applikation fra bilindustrien.

Contents

Chapter 1

Introduction

In the mid 1940s, when the epoch of digital computing started, the first computers were very large and expensive, only available for universities and research centres. Since then, the price and the size of computing systems have been decreasing constantly, and computers become more and more common in our lives. This includes digital watches, CD/DVD players, television, cameras, cell phones, navigation systems, vehicles, aircrafts, and even washers... and many other devices, which we never think of as computers. However, they all do contain a small computer (or many), which is often built for a very specific purpose. We call such computers *embedded computer systems.*

Comparing to personal computers that can be programmed to perform almost any operation, embedded computers are single purpose devices, often restricted by the needs of the application. The application specific implementation allows embedded systems to be faster, more robust and smaller than general purpose PC, but it also makes them more difficult to design. Not only the required functionality has to be implemented, but also factors like production cost, device size, power consumption, performance and fault-tolerance are to be considered very carefully before starting production.

This project is related to a special class of embedded systems, which are called *fault-tolerant embedded systems.* Fault-tolerant systems are used for safety-critical applications, where a single fault might lead to catastrophic consequences, like injuries or loss of human lives and damage to the environment. Such system are typically responsible for critical function control in cars, aircrafts and spacecrafts, nuclear plants, medical devices, etc. They must react to events in the environment within precise time constraints and are therefore called *real-time systems.* Fault-tolerance put very strict requirements on real-time embedded systems, which includes resistance to faults while still meeting any hard deadlines.

Looking at general fault persistence, all faults can be divided into two classes: permanent and transient. Transient faults are induced in hardware, caused by external factors, like radiation particles or lightning stroke that cannot be shielded out. The presence of a transient fault may lead to an error in the application, and this is where the fault-tolerance can be used to save the system from the failure. In this thesis, we address only *transient* faults.

The situation becomes more complex when the system is large and consists of several independent components. Each component is a small embedded computer with CPU, memory and communication capabilities. All components are distributed and interconnected, so they can exchange data in order to work together. We call this type of system a *distributed embedded system.* An example of a distributed fault-tolerant embedded system can be found in a modern car (see Figure 1.1), which contains a lot of safety-critical components: ABS, cruise controller, airbag system, wheel anti-spin etc.

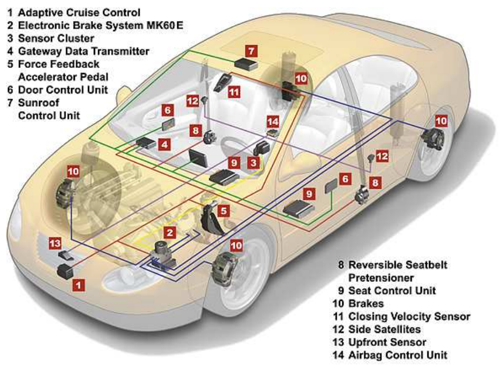

1 Adaptive Cruise Control
2 Electronic Brake System MK60E
3 Sensor Cluster
4 Gateway Data Transmitter
5 Force Feedback Accelerator Pedal
6 Door Control Unit
7 Sunroof Control Unit
8 Reversible Seatbelt Pretensioner
9 Seat Control Unit
10 Brakes
11 Closing Velocity Sensor
12 Side Satellites
13 Upfront Sensor
14 Airbag Control Unit
1
2
3
4
5
6
6
7
8
8
9
9
10
10
10
10
11
12
12
13
14

Figure 1.1: An example of a distributed embedded system [6] with several processing elements and communication busses.

It is the job of the designer to ensure that the embedded system will meet its real-time requirements and produce correct results. As the system can be either time-triggered or event-triggered, the corresponding timing analysis, which checks the timing requirements, will reflect this. When using a time-triggered model, the schedule is determined at design time. For event-triggered systems a static schedule cannot be produced, since the execution depends on external events arriving during the runtime of the system. Recalling the car example the airbag controller could be an event-triggered system, since it executes its programs in reaction to a “collision” event. This thesis focuses on distributed embedded systems with an event-triggered architecture using bus-based communications.

In the following section we will lay out the general design flow of embedded systems, and show where our work is to be applied. Section 1.2 presents the motivation for the thesis. In Section 1.3 we introduce the related work. In the end of this chapter the reader will find a short overview of the structure of the report. The summarised problem formulation is given in Chapter 1.4.

1.1 Design Flow for Embedded Systems

Figure 1.2 shows the system-level design flow for embedded systems. It is based on two inputs, which are the model of the application (software) and the model of the system platform (hardware).

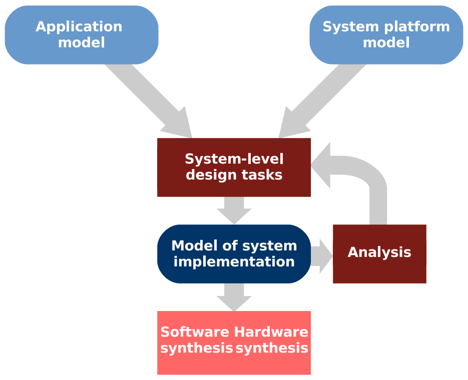


Figure 1.2: The Design Flow for Embedded Systems [30]. This thesis addresses the analysis and the system-level design tasks.

The application model contains processes, including their runtime characteristics, such as deadlines and priorities. The system platform model describes the hardware in the system, which is the embedded devices (computing elements) and the communication channels. The models are used in the stage called *system-level design tasks*, and in this work the related tasks can be listed as below:

- *Application mapping*, this task is about placing the application processes on the different components of the system. For processes it means selecting an appropriate computing element, and for messages it would be selecting communication channels. Some of these mappings might already be decided by the designer.

- *Fault-tolerance policy assignment*, this task is about selecting an appropriate way of protecting the processes against transient faults. Depending on the constraints, each process will be assigned a fault-tolerance technique, such that the timing requirements are met even in case of faults. We consider reexecution and active replication as the techniques to protect against transient faults.

- *Producing a fault-tolerant application* is done when all processes have been mapped and assigned a fault-tolerance policy. The result of this task would be a *model of system implementation*, that describes the execution flow in the system, when transient faults occur. If the produced application satisfies the criteria given by the systems requirements, it can be brought to the next stages.

The tasks above are performed and the results are evaluated in the analysis phase. The analysis of the proposed solution consists of the following parts:

- *Schedulability analysis verifies* that the application can meet the specified timing constraints. In this thesis, we use a response time analysis, which finds the worst case reponse time and then compares it with the corresponding deadline for each process of the application.

- *Performance evaluation* measures the overall application performance, which in our case only includes the response times of the processes. But it may also include other factors, like power consumption, CPU/memory utilisation and so on.

The system-level tasks and the analysis are typically done as an iterative process, consisting of generation-evaluation sequences until a satisfactory solution is found.

1.2 Motivation

With the improvement of manufactoring techniques, the permanent fault rate of systems is decreasing (see Figure 1.3). On the other hand, the number of transient faults is increasing [9]. Today the rate of transient faults is often very large compared to the rate of permanent faults, the ratio varying from 1:4 to 1:1000 according to [2].

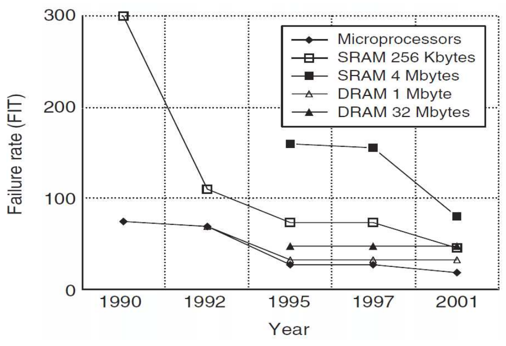


Figure 1.3: Permanent Failure Rates for CMOS Semiconductor Devices [9]

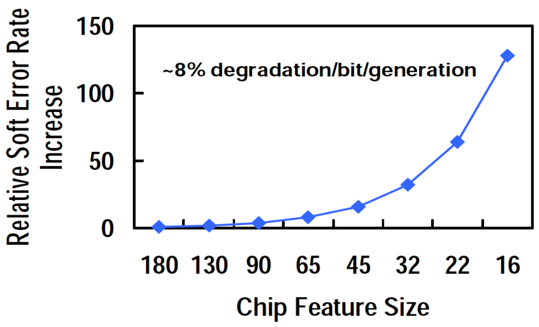


Figure 1.4: Transient Fault Rate Trends [1]

This is, among other things, a result of an increased density of embedded hardware in order to pack more functionality and resources in smaller units. High density leads to increasing electric interference and causes random bit flips. This trend is captured by Figure 1.4. Even in properly designed systems, background radiation and various external electronic magnetic interference will cause transient faults. This problem contradicts the increasing use and dependence on embedded systems, where increasing reliability is a requirement.

Considering the automotive electronics, modern vehicles are integrating more and more electronic devices for providing better control over the vehicle and improved safety. All modern cars have many built-in embedded systems. Furthermore, automotive manufactures are now designing so-called “control-by-wire” systems. Such systems allow more precise control and better performance. On the other hand, they might loose the tangible safety of mechanical components [10]. It has also been observed that car electronics are often affected by transient faults. The electronics in cars are safety-critical embedded systems, and a proper protection against faults is required in order to avoid fatal consequences.

Traditionally, hardware duplication was used to protect critical components against hardware faults. In case of a failure the correctly working components would ensure the stability of the system. Unfortunately this solution is a very costly way of treating transient faults that occur and disappear randomly. And the manufactures need to look at the alternatives, like reexecution and software replication. However, applying reexecution or replication may and probably will introduce significant timing delays in the system, and the designer can possibly end up with a solution that is not schedulable. Therefore, designers may need an optimisation technique that can help them to introduce redundancy in the most cost-efficient way.

1.3 Related Work

Hardware redundancy is a common way of tolerating single permanent faults, and it has been used in a number of fault-tolerant arcitectures, such as MARS[27], TTA[26] and XBW[7]. Hardware redundancy can also be used to protect the components of a system against transient fault, but often such a solution is very impractical due to high cost of the hardware.

There has been academical research that addresses modelling and scheduling of processes on distributed multiprocessor systems [13, 31, 16]. However, many of these assume that the processes are independent, which is not the case in the real world. Processes in an application can have both data and control dependencies. They might need to exchange intermediate results across different platforms and communicate through a medium. The knowledge about process dependencies can be used to improve the accuracy of the response time analysis (RTA). There have been proposed algorithms that can take inter-process dependencies into account [39, 41, 19, 8, 17]. They are all based on the concepts of offset, jitters and phases, which are used to model the time intervals between releases of processes.

Tindell introduced a technique in [39] to compute worst-case response time by using static offsets, and his analysis was later extended and improved in [20] by Palencia and Harbour. In the last paper they developed a better elimination of precedence conflicts and introduced analysis of processes that belong to the same sequence during the execution. The result was a reduced pessimism of the RTA. Their algorithms were called WCDO[1] [19] and WCDOPS[2] [20]. More recently work has been done by Redell, who developed a newer version of the response time analysis with precedence constraints based on the WCDOPS algorithm. Redells algorithm, called WCDOPS+, reuses and improves the concepts defined in [20], which makes it applicable to systems with both preemptive and non-preemptive schedulers. This algorithm has been chosen as the starting point for the response time analysis in this thesis. We also propose several improvements related to precedence and fault conditions.

Different ways of handling both transient and permanent faults have been proposed. Xie et al. in [22] describe an allocation and scheduling algorithm to handle transient faults with replication of critical processes. Very few researchers [21, 29] consider the optimization of implementations to reduce the impact due to fault-tolerance on performance and, even if optimization is considered, it is very limited and does not include the concurrent usage of several redundancy techniques. In [23], Izosimov presents several design optimisation strategies for applying fault-tolerance in embedded systems. More recently Pop et al. propose in [14] a design optimisation approach for statically scheduled applications, using active replication and reexecution as fault tolerance techniques.

1.4 Thesis Objective

The objective of this thesis is to develop and evaluate a design optimisation technique for fault-tolerant embedded systems. We focus on the automatic assignment of fault-tolerance and mapping that protect the system against a fixed number of transient faults while obeying the real-time requirements.

An embedded system in this thesis is described by an application, a hardware architecture and a fault model. The application is a set of processes with possible control and data dependencies. Cyclic dependencies are not allowed. Data dependencies are messages having a sending and a receiving process. A group of processes with mutual dependencies is called a transaction with a period representing the minimum interval between events triggering the transaction.

[1] Worst Case Dynamic Offsets
[2] Worst Case Dynamic Offsets with Priority Schemes

The hardware architecture is distributed and consists of one or more processing elements and possible a number of communication busses. We assume that the scheduling on the processing elements is event-driven, preemptive with fixed priorities, while messages are non-preemptive with fixed priorities and transmitted using a CAN bus. The application contains a mapping table that defines on which processing elements each process can execute. If a process is allowed to run a given processing element, the corresponding best and worst case execution time must be given. The designer must also assign a fixed priority to each process as well as the initial mapping. Messages are statically allocated by the designer to a communication bus. Best and worst case transmission time as well as priority must also be given. Deadlines of processes and messages must be given by the designer and they are always hard. It is assumed that all bounds on execution and transmission time are known prior to the analysis.

The fault model describes how many transitient faults are allowed in every period of the application. In order to tolerate faults, two different approaches are used: reexecution and replication. The dilemma is that both techniques increase the utilisation of computational resources and hence may break real-time requirements to the system.

Therefore, the goals for this project can be stated as following:

- Provide a modelling framework that can represent the system and model the faults.
- Provide a reliable response time analysis that can be used to validate whether the system obeys the timing constraints when considering a fixed number of transitient faults.
- Provide a heuristic algorithm for optimising fault-tolerance assignment and mapping.
- Evaluate and elaborate the proposed methods with synthetically generated systems.
- Evaluate an example from the real world, an adaptive cruise controller.

However, a number of assumptions has been done in the following analysis. These assumptions are basically simplifications of the model:

- The communication system is assumed to be fault-tolerant that is to say we do not protect the messages against transmission errors.

- The overheads related to the system environment are neglected. It means that there are no other applications running on the processing elements except for the one being analysed.
- The operating system is transparent in sense that all overheads are included in the execution times of the processes
- Deadlines of processes must be shorter than the period of the transaction
- When doing reponse time analysis of fault-tolerant process graphs, only one transaction is allowed

1.5 Thesis Overview

In the following we will briefly present the different parts of this thesis and explain the relations between the corresponding chapters. We have divided the thesis into a number of sub problems as shown Figure 1.5. The arrows in the figure are used to illustrate the relations between the defined sub problems.

To begin with, all necessary basic theoretical concepts will be explained in Chapter 2. This preliminary chapter contains an introduction to the hardware, application and fault models used in the thesis, presents fault tolerance techniques and the basics of scheduling.

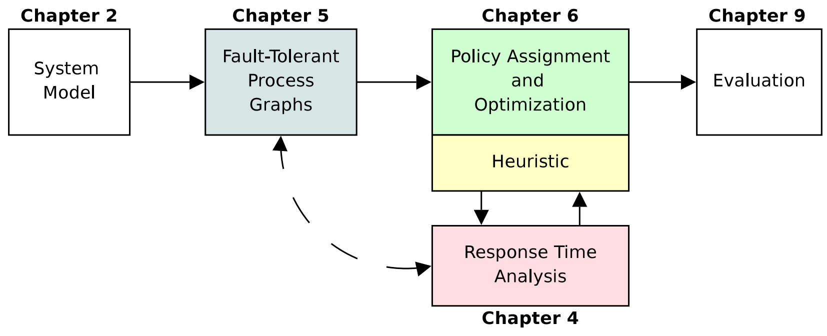


Figure 1.5: Thesis Guideline Diagram

Chapter 3 addresses the response time analysis. It contains a description of the existing WCDOP+ algorithm, including all necessary details to understand how it works. Besides that, the chapter includes a complete description of the extensions proposed to the algorithm. Chapter 4 deals with fault-tolerant process graphs and covers the basic notation and definition of elements. It describes the data structures and necessary transformations applied when changing between different fault-tolerance techniques and mappings.

Chapter 5 is entirely dedicated to the fault-tolerance assignment and optimisation. It explains how to choose the most optimal policy assignment using the response time analysis described before. With this theoretical foundation in place, we give some details on our implementation in Chapter 6 including data structures, relations between equations and methods, and the pseudocode for some of the operations on our data structures. The optimisations we have done to improve the performance of the program will also be discussed. In this chapter we will also explain our approach to testing, which includes both unit tests and functional tests.

In Chapter 7 an extensive evaluation of our algorithm will be performed and discussed. These evaluations are done on numerous synthetic applications and a cruise controller example from the automotive industry. We summarise and make conclusions on the work in relation to the obtained results in Chapter 8. Based on our work, suggestions for future work are listed and discussed in Section 8.1.

Notice that Appendicies A and B contain lists of abbreviations and notations, respectively, that will be used throughout the report.

CHAPTER 2

Preliminaries

In this chapter we will lay out the preliminaries for the thesis. This includes notations, model description and an introduction to fault tolerance related to this thesis. In Section 2.1 we give a comprehensive description of the system model, which includes both description of the hardware and the software architectures. Section 2.2 describes faults in the given context and presents several related fault-tolerance techniques, including process replication and reexecution.

2.1 System Model

This section explains the system model, which includes the hardware platform and application model, i.e. the software to be executed on the hardware platform.

2.1.1 Hardware Architecture

The hardware architecture of the embedded system is composed of a set of processing elements, which are distributed and interconnected by one or more communication channels, see Figure 2.1. Each processing element consists of a

CPU, memory and a communication subsystem. The communication system is responsible for low-level operations, such as communication protocols and error correction during communication.

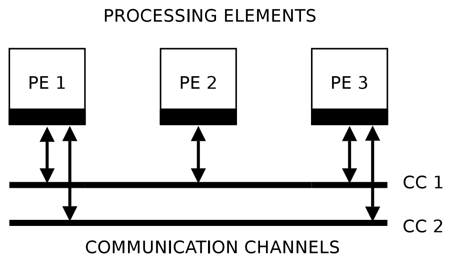


Figure 2.1: A View of the Hardware Architecture. The shown system contains three processing elements connected by two communication channels.

In the thesis we handle all subsystems on the processing elements as a whole, regardless of any overheads that may be introduced by interaction between different hardware levels. The size of communication buffers and memory are, for simplicity, not considered.

The messaging subsystem contains an arbitrary number of communication channels that are used to deliver messages between the processes. There exists a number of busses and protocols for transferring data between the processes. Some are general purpose while others are specific for a particular industry. As we are only considering event-triggered systems and since the project relates to the automotive industry, we have assumed that Controller Area Network (CAN) is used for the communication. We make the simplification and assume that no faults will happen during communication or they will be tolerated using existing techniques. Figure 2.2 shows a CAN bus connecting different subsystems in a modern car.

Messages being sent over the CAN are not preemptive. Once the transmission of a given message has started, other processing elements cannot start a new transfer before the transmission has finished. This also implies that only a single message can be transmitted at any given time on a particular communication channel. However, since CAN defines message content rather than message destination, the same message will be received by all processing elements on the bus. The messages do also have priorities that can be used to determine what message to send, if more than one message are ready to be sent. This is implemented by the arbitration field of the CAN frames.

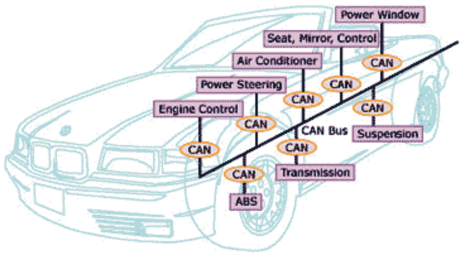


Figure 2.2: An example of a Controller Area Network (CAN) in a car [11] where different components are connected through the bus.

If two processes are mapped on the same processing element, then the transmission time of a message is neglected. In this case the message will be placed directly into the shared memory, when the sending process finishes. In a situation when the sending and receiving processes are not on the same processing element the message will be sent through the communication channel.

2.1.2 Software Architecture

The software architecture used on the top of the hardware is a real-time operating system (RTOS), which can perform in such a way that all timing requirements are satisfied. As we are only interested in the real-time properties of the RTOS, all other details are omitted being irrelevant.

The execution model is based on real-time preemptive scheduling with fixed priorities. It means that the RTOS can switch between processes, and processes having higher priorities will interrupt execution of lower priority processes. When it happens, the lower priority process has to wait until the high priority process finishes its execution. Priorities are related to the process importance, and are given by the designer. Preemption can happen at any point of time, as it depends on the priorities of the processes and even arrivals.

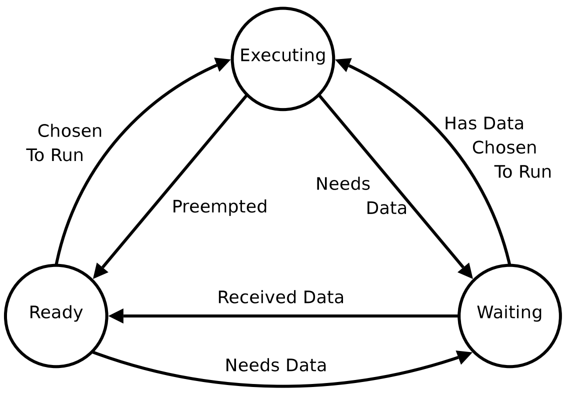


Figure 2.3: Scheduling States of a Process. The diagram shows how the RTOS controls the execution of a process.

The scheduling states of a process follow the scheme proposed at [40] and is shown in Figure 2.3. If several processes are active, i.e. ready to be run, the schedule will always choose the one with the highest priority. If those processes have the same priority, the choice will be non-deterministic.

Generalisation of the RTOS running on top of each processing element is done with the assumption that the following overheads are included in the worst case execution time of the processes:

- Context switch and process activation.
- Error detection.
- Recovering of process inputs.
- Interaction with the communication layer.

We also assume that some synchronisation takes place between the processing elements achived through message transmissions. It is simply assumed that the mechanism is given and does not introduce any overhead.

2.1.3 Application Model

The application model describes a set of processes that together form an *application*, denoted $\mathcal{A}_i$. An application is represented as a set of acyclic directed graphs, $\mathcal{A}_i = \Gamma_a(\mathcal{V}, \mathcal{E})$. An example is shown in Figure 2.4.

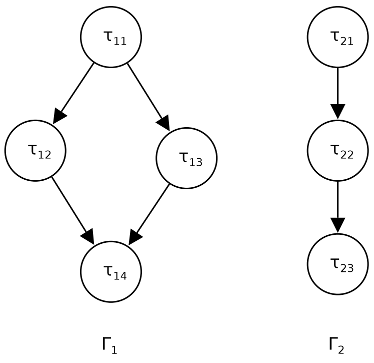


Figure 2.4: An Example of an Application Consisting of Two Transactions

A graph Γ_a is also called a *transaction* or *process graph* and has a period T_a. Each vertex $\tau_a \in \mathcal{V}$ in the graph Γ_a represents a process, and each edge $e_{ij} \in \mathcal{E}$ from τ_i to τ_j represents a precedence constraint between two processes. By using precedence constraints we can model the order of execution of the processes in an application. It means that a process having precedence constraints from other processes, cannot be executed before all of those processes have finished even if it has a higher priority. A transaction therefore groups processes that have precedence constraints. A process that have no precedence constraints is called the *root process*.

Each precedence constraint between two processes may have an associated *message*, m_i, sent through one of the communication channels. In this case, the precedence relation is called a *data dependency*. A message is only transferred when the sending process has finished, and the receiving process cannot start its execution before the message has been completely received.

Figure 2.5 shows a single transaction consisting of five processes (τ_1, ..., τ_5), illustrating the graph representation of an application. In the figure, the precedence constraints are drawn as edges without messages, whereas the messages are drawn using boxes m_1 and m_2 on the edges.

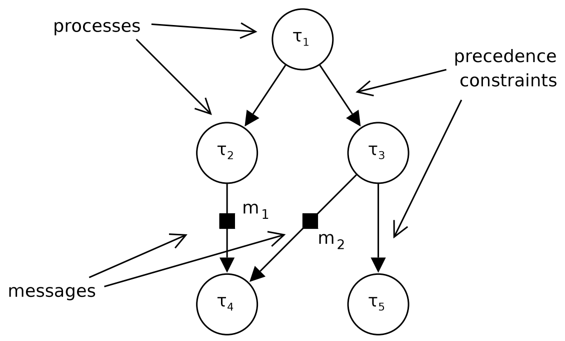


Figure 2.5: An Example of a Process Graph. The arrows show the elements of the graph.

A process is described by a set of temporal and execution properties, given in Table 2.1.

Notation	Short Description
C_a	Worst Case Execution Time (WCET)
C_a^b	Best Case Execution Time (BCET)
P_a	Priority
D_a	Deadline
T_i	Period

Table 2.1: Properties of a Process

The execution times are the lower and upper bounds of time required for a process to complete. In the model, execution time depends on the chosen processing element. Therefore the execution times are given as a table, where each pair of process and processing element is represented by best and worst case execution times. Such a table is called a *mapping table* and an example is given in Table 2.2. If a mapping is not allowed, the corresponding entry in the table is empty. In Table 2.2, process τ_2 is not allowed to execute on processing element N_2.

	N_1	N_2
τ_1	(1,2)	(2,3)
τ_2	(7,7)	
τ_3	(6,9)	(7,10)

Table 2.2: Example of a mapping table from a system model containing the best- and worst case execution time for the processes on the different processing elements.

The priority indicates the importance of the process – higher numbers mean higher priority, so the execution of low priority processes may be preempted by higher priority processes. The deadline is the latest point in time, at which the process is supposed to have finished executing. In real-time systems with hard deadlines all processes must successfully complete before their deadlines. In this model, the deadlines are absolute, meaning that the time is counted from the arrival of the event triggering the execution. The period of the transaction represents the minimum interval of time between any two events causing activation of the transaction. Deadlines are not allowed to be longer than the period of the transaction. Because of the precedence relations defined by the transaction, the period for all processes in the given transaction must be equal to the period of the transaction.

Similarly to processes, messages are characterised by the following properties:

Notation	Short Description
C_i^m	Worst Case Transmission Time (WCTT)
C_i^{mb}	Best Case Transmission Time (BCTT)
P_i^m	Priority
D_{ij}^m	Deadline

Table 2.3: The properties of a message.

All messages are statically assigned to a communication channel, and this cannot be changed during the design optimisation.

2.2 Fault Model

As mentioned in the previous sections, we do not look at the fault detection. We assume that the RTOS contains a mechanism to detect the faults and notify the scheduler. The time needed for the detection of an error is called error detection overhead, and the time needed for the system to restore the initial state of the process is called recovery overhead. We have assumed that these overheads are included into process execution times, and state recovering takes no time. We denote the maximum number of transient faults that might happen during one period of the transaction as κ.

When a fault occurs, we always assume that it happens at the worst possible instant in time. This is exactly when the process is about to finish, thereby introducing the maximum delay for subsequent processes.

We are now going to present two fault-tolerance techniques, which are relevant for our model. These are reexecution and replication, which are the most used techniques for tolerating transient faults. Notice that a process can only be protected by one of the two techniques.

2.2.1 Process Reexecution

Reexecution provides fault-tolerance by running a process a second time on the same processing element if it fails. Indeed the reexecution may be one of the natural ways of dealing with faults - if something did not work, then try one more time. The use of reexecution may be unsuitable in some situations, because the successful completetion will be significantly delayed.

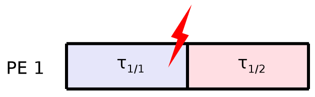
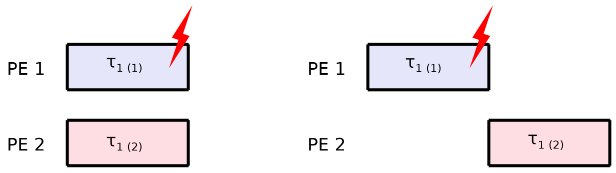


Figure 2.6: An Example of Reexecution. Process $\tau_{1/2}$ is only run, when process τ_1 fails.

Consider Figure 2.6 where process τ_1 fails and is reexecuted. The j^{th} execution of a process is denoted by slash in subscript, so the first execution is written as $\tau_{1/1}$ and the first reexecution is written as $\tau_{1/2}$. The set of processes protected with reexecution is denoted $\mathcal{P}_x$. The reexecution approach has the advantage that it is simple to implement. On the other hand the reexecution approach may prolong the response time of the faulty process and thus may result in missed deadlines.

2.2.2 Process Replication

Another way of protecting processes against faults is to use process replication. Compared to the reexecution, which is based on time redundancy, the replication approach uses space redundancy. When using *active* replication, several instances of the same process, called *replicas*, are executed in parallel on different processing elements independently of fault occurrences. With *passive* replication, replicas will only start, if the primary process fails. Both methods are illustrated in Figure 2.7. In our thesis we focus only on active replication.



(a) Active replication. (b) Passive replication.

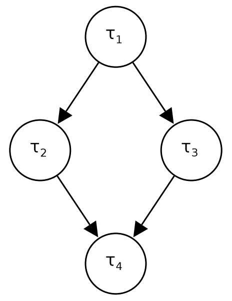
Figure 2.7: Two Types of Process Replication.

We denote replication with round brackets in subscripts surrounding the replica number, j. For the main process the replica number is always zero, and its replicas will have consecutive numbers starting from one. In the figure above the primary process is denoted as $\tau_{1(0)}$, and its replica is $\tau_{1(1)}$. The set of processes that are protected with replication, is denoted $\mathcal{P}_r$.

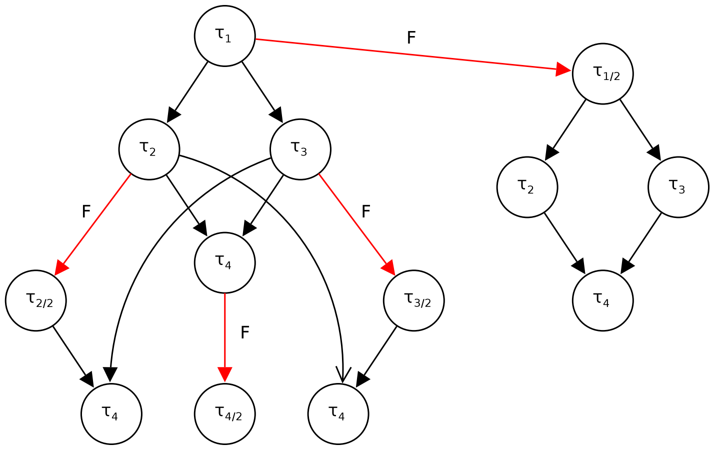
The main advantage of active replication is that a transient fault does not delay the response time of the protected process as much as with reexecution in most situations. On the other hand, the system will also have to execute the replicated process if no faults occur at all, and therefore consume more resources.

2.2.3 Fault-Tolerant Process Graph

In order to model fault occurences in our system we use conditional process graphs (FTPG), denoted $\mathcal{G}_a$. A conditional process graph is similar to a regular process graph, extended by adding guards on the edges to model fault occurences. The guards are boolean conditions indicating the presence of a fault.



(a) Original process graph



(b) Fault-tolerant process graph

Figure 2.8: Producing Fault-Tolerant Process Graph with All Processes Being Reexecuted.

Figure 2.8(a) demonstrates a process graph, that has been extended to a fault-tolerant process graph in Figure 2.8(b). All processes are set to be reexecuted in case of faults, and the maximum number of tolerated faults κ is 1. Depending on the presence of fault for a process, the corresponding edge must be taken. If a process finishes successfully, then all non-faulty edges must be taken. If the process fails, then it must be reexecuted and therefore the execution path will include the conditional edge, F, starting at the faulty process. The fault and non-fault conditions are mutually exclusive for a given process, and only one type can be taken during the execution.

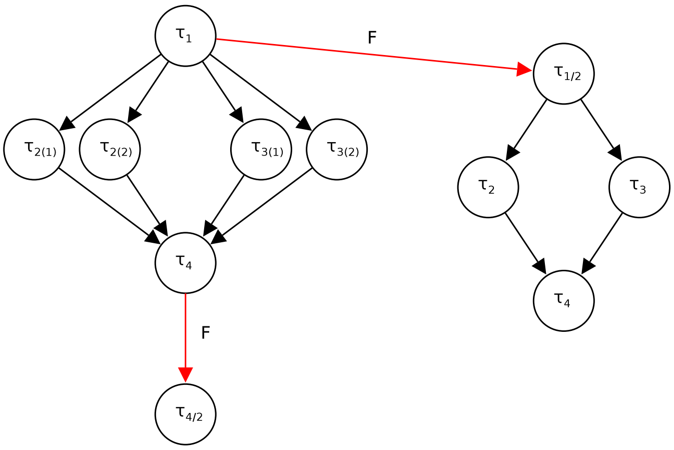


Figure 2.9: Combining Reexecution and Replication. Processes τ_2 and τ_3 are replicated, while processes τ_1 and τ_4 are reexecuted

Figure 2.9 shows how the reexecution and the replication can be combined when $\kappa = 1$. The shown graph represents a fault-tolerance policy assignment by which processes τ_1 and τ_4 are chosen to be reexecuted, and processes τ_2 and τ_3 are protected with active replication. It should be noted, that if process τ_1 fails and is reexecuted as $\tau_{1/2}$, the succeeding processes will not experience any faults (recall that $\kappa = 1$). It leads to removal of replicas $\tau_{2(2)}$, $\tau_{3(2)}$ and reexecution $\tau_{4/2}$ from the fault-scenario started by the fault in τ_1.

As mentioned earlier, we do not combine the reexecution and replication for the same process. When using replication, we always assume no faults have occured during the execution of the replicas. Therefore we need to protect succeeding processes against the same number of faults as the process being replicated. In contrast, reexecution captures the presence of a fault, and the succeeding processes must be protected against $\kappa - 1$ faults.

A specific trace or execution path through an FTPG for a certain combination of faults captured, is called a *fault scenario* and is denoted s_{is}. Any FTPG will always have at least one fault scenario - the situation with no faults. The set of all fault scenarios for a given FTPG, $\mathcal{S}_i = \{\forall \mathrm{s}_{is} \in \mathcal{G}_i\}$, represents all possible combinations of faults that can be captured. It includes also situations with less than κ faults.

We now have presented the basic terminology and the models that will be used through this thesis. On this foundation, we will define the objective of this thesis more closely in the following chapter.

Chapter 3

Response Time Analysis

In this chapter we present the response time analysis algorithm for event-triggered systems, WCDOPS+, which we use as a starting point for our schedulability analysis. The original version was developed by Ola Redell and described in his works [35, 34]. We have extended the WCDOPS+ in order to deal with fault-tolerant scheduling. The basic idea of response time analysis (RTA) is to determine the worst-case response times of all processes. By comparing the worst case response times with the deadlines, we can test the schedulability of the system.

The chapter is structured as follows. Section 3.1 explains the approach used in WCDOPS+ and introduces the theoretical background required to understand how the algorithm works. Then we will present our extensions to the algorithm in Section 3.2, and explain how the algorithm can be applied on fault-tolerant applications in Section 3.3.

Details on the implementation of the algorithm are described in Chapter 6.

3.1 Basic WCDOPS+

In this section we will describe the basic WCDOPS+ analysis as given in [34]. All equations, unless otherwise stated, are taken from [34].

The WCDOPS+ algorithm allows us to perform response time analysis on distributed event-triggered systems. The processes are grouped into transactions. The execution of a transaction, Γ_i, is triggered by an external event. The events arrive aperiodically with a minimum interval between the releases denoted by T_i. Each transaction contains a set of processes having precedence, which form a tree-shaped acyclic graph, as shown in Figure 3.1. An activation of the transaction Γ_i is called a *job*, implying that all processes of the transaction will belong to the same job for the arrival of a given event.

The processes are identified by two subscripts: an unique number among other processes in the same transaction and the number of transaction, they belong to. A transaction has only *one* root process, and in Figure 3.1 the root process is τ_1. The processes are mapped on different processing elements, and the mapping is given by $\mathcal{M}(\tau_{ij})$. The priority of a process is denoted by P_{ij}. For the best case response time of a process τ_i we use the notation R_i^b, and the worst case response time is denoted by R_i^w.

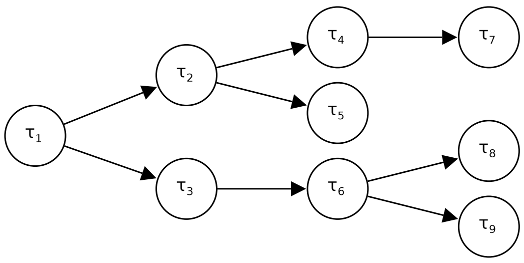


Figure 3.1: An Example of a Tree-Shaped Transaction with Nine Processes

3.1.1 Modelling Precedence Relations

The precedence relations are expressed by offsets and jitters, which help to model processes with precedence as they were independent. An offset, Φ_{ij}, is the minimum relative time after an event has arrived to the activation of process τ_{ij}. The offset is then the earliest possible instant at which a process can start executing. The jitter, J_{ij}, is the maximum delay that a process can experience from its earliest possible arrival until it is released. Then, if the event arrives at time t_0, the latest point in time process τ_{ij} can be released, is given by $t_0+\Phi_{ij}+J_{ij}$. Figure 3.2 illustrates the relation between the arrival of the event and the execution of process τ_{ij}.

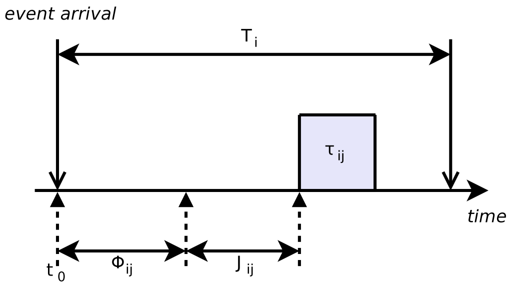


Figure 3.2: Offset and Jitter Relation

Offsets and jitters are dynamically updated between the iterations of the algorithm as follows:

$$\Phi_{ij} = R^b_{ip} \tag{3.1}$$

$$J_{ij} = max(R^w_{ip} - R^b_{ip}, J_{ip}) \tag{3.2}$$

The equations show that the offset is found as the best case response time of the preceding process τ_{ip}, and the jitter is the difference between worst case and best case response times of the preceding process.

The WCDOPS+ algorithm will find best case and worst case response times for each process in the system. The particular process, which is currently being analysed, is denoted as τ_{ab}. The main idea behind the analysis is to find the maximum possible interference from other processes in the system, that may delay the execution of process τ_{ab} either due to preemption or precedence relation. The maximum interference is found by analysing the *busy period* of τ_{ab}. The busy period is the period, when the processor on which τ_{ab} is mapped, is occupied by other processes having the same or higher priority as τ_{ab}. This implies that the execution of τ_{ab} will be preempted by these processes. The set of processes that run on the same processing element as τ_{ab} and having equal or higher priority is given by $hp_i(\tau_{ab})$. Consequently $lp_i(\tau_{ab})$ represents the set of lower-priority processes. Formally $hp_i(\tau_{ab})$ is defined as

$$hp_i(\tau_{ab}) = \{\tau_{ik} \in \Gamma_i \mid P_{ik} \geq P_{ab} \wedge \mathcal{M}(\tau_{ij}) = \mathcal{M}(\tau_{ab})\} \tag{3.3}$$

and $lp_i(\tau_{ab})$ is given by

$$lp_i(\tau_{ab}) = \{\tau_{ik} \in \Gamma_i \mid P_{ik} < P_{ab} \wedge \mathcal{M}(\tau_{ij}) = \mathcal{M}(\tau_{ab})\} \tag{3.4}$$

These are very important definitions and will be used extensively through the analysis.

3.1.2 Process Phasing

The busy period of τ_{ab} starts at some point of time, called the *critical instant* t_c. The worst case delay for τ_{ab} will be created, when processes in $hp_i(\tau_{ab})$ are phased in such a way that they are released at t_c. In this situation, the execution of τ_{ab} will be delayed the most due to these higher-priority processes. The algorithm also takes into account the interference from other transactions in the application, that might have different periods, than the transaction to which τ_{ab} belongs. The maximum contribution to τ_{ab} busy period from a given transaction Γ_i happens when a process τ_{ik} in $hp_i(\tau_{ab})$ originating from Γ_i starts the busy period. However, in case when deadlines are larger than event periods, the interference can also occur from previous jobs of the same transaction.

In order to find the maximum contribution from a particular transaction Γ_i, the algorithm must identify all processes from all jobs of the transaction that are ready to be executed in the busy period. To identify all pending instances of a process τ_{ij} a phase relation φ_{ijk} between τ_{ij} and τ_{ik} is used to find the earliest possible arrival of process τ_{ij} after t_c. The phase is defined as

$$\varphi_{ijk} = T_i - \Phi_{ij} - (\Phi_{ik} + J_{ik}) \bmod T_i \tag{3.5}$$

and the total number of pending instances n_{ijk} of process τ_{ij} at t_c is

$$n_{ijk} = \left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor \tag{3.6}$$

The jobs and the corresponding instances of processes are assigned an index p, based on the arrival time of the external event relative to t_c. The positive values are assigned to instances coming after t_c, whereas 0^{th} and negative indexes indicates that the job arrives prior to t_c. The phase relation and numbering of pending jobs is illustrated in Figure 3.3.

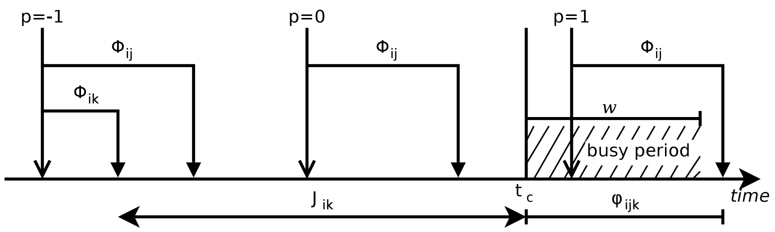


Figure 3.3: Job Numbering and Phasing for Process τ_{ij} relative to t_c

The *width of the busy period* started by τ_{ik} at t_c is given by w, and the contribution from a process τ_{ij} is only possible, if the processes are phased in such a way, that τ_{ij} is released during w. In Figure 3.3 the number of pending instances n_{ijk} of τ_{ij} is therefore 2, with indexes $p = -1$ and $p = 0$. The latest instance has always index $p = 0$, and thus the index p of the first pending instance of τ_{ij} can be found as

$$p_{0,ijk} = 1 - n_{ijk} = 1 - \left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor \tag{3.7}$$

which in our case is $p = -1$, i.e. triggered by the earliest event arrival.

3.1.3 Process Grouping

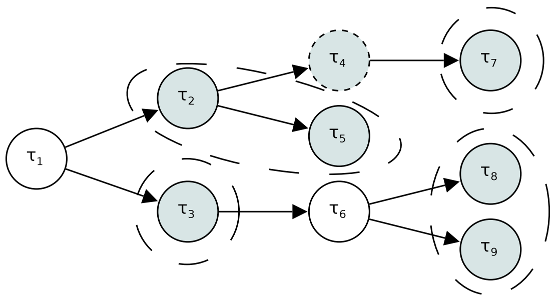
Another important concept, which is presented in the WCDOPS+ algorithm, is grouping of processes in *H-sections* and *H-segments*. Sections and segments consist of many processes, that due to their priorities and precedence relations can be treated as a single large process. They are used in the analysis, since they group processes that may belong to the same busy period of process τ_{ab}. Segments and sections are always determined *relative to a given process*, which in our case is τ_{ab}. The priority and mapping of τ_{ab} is used to check whether two processes are in the same segment or section. The definition of an H-segment is given as follows:

$$H_{ij}^{seg}(\tau_{ab}) = \{\tau_{ij} \in hp_i(\tau_{ab}) \mid (\neg\exists\tau_{il} \in \Gamma_{ij}\Delta\Gamma_{ik} \mid \tau_{il} \notin hp_i(\tau_{ab}))\} \tag{3.8}$$

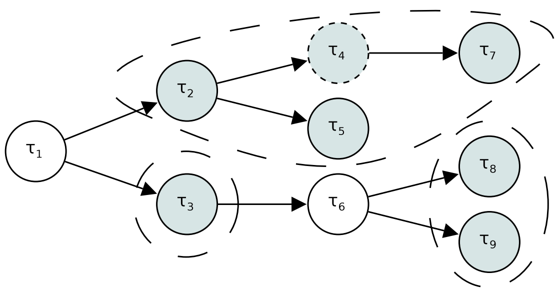
Two processes that are in $hp_i(\tau_{ab})$ belong to the same segment, if there is no other processes $\tau_{ij} \notin hp_i(\tau_{ab})$, that precedes one process but not the other process. In other words, the processes in a segment may not contain any intermediate processes that are not in $hp_i(\tau_{ab})$. The main property of an H-segment is that if one process from the segment belongs to the busy period, then *all processes in the segment will contribute to the busy period*, and this is why a segment may be considered as a single large process. In contrast to H-segments, H-sections group processes, that *may belong to the same busy period*. An H-section $H_{ij}(\tau_{ab})$ is defined as follows:

$$H_{ij}(\tau_{ab}) = \{\tau_{ij} \in hp_i(\tau_{ab}) \mid (\neg\exists\tau_{il} \in \Gamma_{ij}\Delta\Gamma_{ik} \mid \tau_{il} \in lp_i(\tau_{ab}))\} \tag{3.9}$$

The equation is similar to equation (3.8), with the difference that two processes in the same section must be preceded by the same process from $lp_i(\tau_{ab})$. This implies, that the processes in an H-section can be interconnected by some intermediate processes, that run on other reprocessing elements. In the following, we will always assume that the priority of segments and sections are given by τ_{ab}. Therefore we will use the shorter notations, H_{ij}^{seg} and H_{ij}, where τ_{ab} is implied.



(a) Segments



(b) Sections

Figure 3.4: Examples of Segments and Sections

The example shown in Figure 3.4 will explain how H-sections and H-segments may look. Processes in Figure 3.4 are painted with fill color according to their priorities. The dark nodes on the graph mark the processes that are in $hp_i(\tau_{ab})$, the white nodes are processes in $lp_i(\tau_{ab})$, and the dashed node τ_4 represents a process, which runs on other processing element than τ_{ab}. Using the definitions, we find four segments and three sections. The segments are $H_{i2}^{seg} = H_{i5}^{seg} =$

$\{\tau_2,\tau_5\}$, $H_{i3}^{seg} = \{\tau_3\}$, $H_{i7}^{seg} = \{\tau_7\}$, $H_{i8}^{seg} = H_{i9}^{seg} = \{\tau_8,\tau_9\}$. The sections shown on in the figure are $H_{i2} = H_{i4} = H_{i5} = H_{i7} = \{\tau_2,\tau_5,\tau_7\}$, $H_{i3} = \{\tau_3\}$ and $H_{i8} = H_{i9} = \{\tau_8,\tau_9\}$.

An H-segment is preceded by a process, $\tau_{ip} < H_{ij}^{seg}$, when τ_{ab} precedes all processes in the segment. In the example above process τ_1 precedes all segments in Γ_i, $\tau_6 < H_8^{seg}$ and $\tau_4 < H_7^{seg}$. A process is said to be an immediate predecessor of a segment, if one of its immediate successors belong to the segment. Processes, with an immediate predecessor in the segment, are called successors of the segment. The precedence properties are mostly important for H-segments, but they are defined in similar way for H-sections as well.

The analysis will study the contribution from all possible segments to the busy period. Similarly to the process phasing, it defines phasing relations between H-segments and the busy period, which are very close to equations (3.5) - (3.7). The phasing of an H-segment is determined by its offset and jitter, which are equal to the first process in the segment. The time from the start of the busy period to the arrival of the segment is given by:

$$\varphi_{ijk}^{seg}(\tau_{ab}) = T_i - \Phi_{ij}^{seg}(\tau_{ab}) - (\Phi_{ij} + J_{ij}) \mod T_i \tag{3.10}$$

and the number of the first pending instance of the segment at t_c is:

$$p_{0,ijk}^{seg}(\tau_{ab}) = 1 - \left\lfloor \frac{J_{ij}^{seg}(\tau_{ab}) - \varphi_{ijk}^{seg}(\tau_{ab})}{T_i} \right\rfloor \tag{3.11}$$

The analysis can be further refined by the fact that some segments can block execution of other segments. Such segments are called *blocking segments*. Only one blocking segment among all blocking segments can execute within the busy period of τ_{ab}. An H-segment is blocking when it has predecessors that belong to $lp_i(\tau_{ab})$. In Figure 3.4 (a) the blocking segments are H_{i2}^{seg}, H_{i3}^{seg} and H_{i8}^{seg}. We return to the segments later, when we present our extensions. Now, the next step is to explain how the worst-case response time is computed by WCDOPS+.

3.1.4 Identifying the Contributions From Other Processes

By using phasing it is possible to find all combinations of processes and segments, that may contribute to τ_{ab} busy period. The contribution is found from each transaction, including the one, process τ_{ab} belongs to. For each transaction two kinds of contribution are computed - *non-blocking interference* W_i, and *blocking interference* WB_i. The non-blocking interference is the maximum contribution from transaction Γ_i, when no blocking segments are allowed to execute within the busy period, and the blocking interference is the maximum contribution from Γ_i when one blocking segment is allowed to execute within the busy period. The difference between blocking and non-blocking interference is called *interference increase* $\Delta W_i = WB_i - W_i$, and the interference increase is maximised among all transactions.

The transaction interference is found by using the function called *Transaction-Interference*, that locates the contribution from process instances arriving prior to t_c ($p \leq 0$). It considers all jobs of a transaction Γ_i that can possible interfere with τ_{ab} in order to locate the worst interference. For each job, it uses the function *BranchInterference* to locate the longest[1] possible chain of higher priority processes that might contribute to the busy period. As not all processes of a given chain are able to actually contribute to the busy period, *TaskInterference* will eliminate these processes during the analysis. This is done by using so-called *reduction rules*, which are simple conditions applied in *TaskInterference.* We refer to [34] and [35], where these rules are explained and formalised and the pseudocode is given. A more detailed description of *BranchInterference* will be given in Section 3.2.3, where we present our modifications to the algorithm.

As mentioned before, *TransactionInterference* finds two values for transaction contribution to the busy period, the blocking and non-blocking interference. However, we also need to find the contribution from instances, that might arrive after t_c ($p > 0$). Due to precedence order, only instances of those processes that belong to the first H-section in the transaction can contribute to τ_{ab} busy period. This requires that the first H-segment is not blocking. Those processes are found as follows:

$$MP_i(ab) = \{\tau_{il} \in hp_i(\tau_{ab}) \mid (\neg\exists \tau_{ix} \in lp_i(\tau_{ab}) \mid \tau_{ix} < \tau_{il})\} \tag{3.12}$$

The contributions from jobs arriving after t_c is then

$$W_{ik}(\tau_{ab}, w)\mid_{p>0} = \sum_{\tau_{ij} \in MP_i(ab)} \left\lceil \frac{w - \varphi_{ijk}^{seg}(\tau_{ab})}{T_i} \right\rceil \tag{3.13}$$

Finally, the total contribution from a transaction Γ_i to the busy period of τ_{ab} when process τ_{ik} is used to start the busy period is given by

$$\begin{aligned}[W_{ik}(\tau_{ab}, w), WB_{ik}(\tau_{ab}, w)] &= TransactionInterference(\tau_{ab}, \tau_{ik}, w) \\ &\quad + W_{ik}(\tau_{ab}, w)\mid_{p>0} \end{aligned} \tag{3.14}$$

But since there can be many processes that can start the busy period, all of them must be considered in order to find the upper bound on the blocking and

[1]In terms of execution time

non-blocking interference and the largest interference increase:

$$\begin{aligned} W_i{}^*(\tau_{ab}, w) &= \max_{\forall \tau_{ab} \in XP_i(\tau_{ab})} W_{ik}(\tau_{ab}, w) & (3.15) \\ WB_i{}^*(\tau_{ab}, w) &= \max_{\forall \tau_{ab} \in XP_i(\tau_{ab})} WB_{ik}(\tau_{ab}, w) & (3.16) \\ \Delta W_i{}^*(\tau_{ab}, w) &= WB_i{}^*(\tau_{ab}, w) - W_i{}^*(\tau_{ab}, w) & (3.17) \end{aligned}$$

The set XP_i used in (3.15) and (3.16) contains all processes in the transaction Γ_i, that come first in their H-segments. The contribution from the transaction Γ_a, which process τ_{ab} belongs to, is found separately, but in a similar way.

3.1.5 Deriving the Response Times

It is now possible to explain how the worst case response time of process τ_{ab} is computed. The analysis is done for all instances of process τ_{ab}. For a single instance the *completion time* consists of following parts: the maximum blocking from low priority processes B_{ab}, (ignored in this thesis), the non-blocking interference from transaction Γ_a, the sum of non-blocking interferences from all other transactions and the maximum blocking interference increase $\Delta W_{ac}{}^*$, due to one blocking H-segment among all transactions. The completion time of p_{ab} is given by this equation:

$$\begin{aligned} \mathrm{w}_{abc}(p_{ab}) &= B_{ab} + W_{ac}(\tau_{ab}, w, p_{ab}) \\ &\quad + \sum_{\forall i \neq a} W_i{}^*(\tau_{ab}, w, \tau_{ac}) \\ &\quad + \Delta W_{ac}{}^*(\tau_{ab}, w, \tau_{ac}) \qquad (3.18) \end{aligned}$$

The response time of instance p_{ab} is found by subtracting the arrival time of the instance from the completion time, w_{abc}, and adding the offset, Φ_{ab}. The subtraction will reduce the completion time to the amount that overlaps with the busy period:

$$R^w_{abc}(p_{ab}) = (\mathrm{w}_{abc}(p_{ab}) - (\varphi_{abc} + (p_{ab} - 1)T_a)) + \Phi_{ab} \qquad (3.19)$$

Notice that the first part of the equation without adding the offset is called *local response time.* The *global response times* include offsets, and they are computed when local response times have been found for all processes. The number of instances of τ_{ab} can be found, when we know the maximum length of the busy period of τ_{ab}, as previously shown in Figure 3.3. The upper bound for the length of the busy period of τ_{ab}, L_{abc}, is computed as follows:

$$\begin{aligned} L_{abc} &= B_{ab} + W_{ac}{}' + \sum_{\forall i \neq a} W_i{}^*(\tau_{ab}, L, \tau_{ac}) \\ &+ \max(WB_{ac}{}' - W_{ac}{}', \Delta W_i{}^*(\tau_{ab}, L, \tau_{ac})) \end{aligned} \tag{3.20}$$

The length of the busy period is used to find the latest instance of the H-segment $H_{ac}^{seg}(\tau_{ab})$

$$p_{L,abc}^{seg}(\tau_{ab}) = \left\lceil \frac{L_{abc} - \varphi_{0,abc}^{seg}(\tau_{ab})}{T_a} \right\rceil_0 \tag{3.21}$$

so the possible instance numbers of τ_{ab} are included in the interval from $p_{0,abc}^{seg}(\tau_{ab})$ to $p_{L,abc}^{seg}(\tau_{ab})$. The final worst-case response time of process τ_{ab} is the response time of the instance having the largest response time, maximised over all possible combinations with processes that may start the busy period:

$$R_{ab}^{w} = \max_{\forall \tau_{ac} \in XP_a(\tau_{ab})} \left[\max_{p_{ab} = p_{0,abc}^{seg}(\tau_{ab}) \ldots p_{L,abc}^{seg}(\tau_{ab})} R_{abc}^{w}(p_{ab}) \right] \tag{3.22}$$

The equations above are solved by using *fixed-point iteration*, and by applying equation (3.22) to all processes in the system, the local response times are found. When response times have been found for all processes, the algorithm updates the offsets and jitters by using formulas (3.1) and (3.2). A simple pseudocode illustrating the outer loop of the algorithm is shown below:

Algorithm 1 Outer Loop of WCDOPS+

```
initLocalResponseTimes($\mathcal{A}$)
repeat
  for all $\Gamma_i \in \mathcal{A}$ do
    for all $\tau_{ab} \in \Gamma_i$ do
      findLocalResponseTimes($\tau_{ab}$) {Equation (3.22)}
    end for
  end for
  for all $\Gamma_i \in \mathcal{A}$ do
    updateGlobalResponseTimes($\Gamma_i$)
    updateJitterAndOffset($\Gamma_i$) {Equations (3.1) - (3.2)}
  end for
until converged
```

When WCDOPS+ detects that there are no changes in the response times, it stops, and the analysis is said to have converged.

3.1.6 Messages and Non-Preemptive Processes

The analysis also includes support for non-preemptive scheduling. It allows us to do response time analysis of the communication on a CAN bus by treating messages as non-preemtive processes. Each communication channel will be represented as a pseudo processing element. Consequently, equation (3.18) is used to find the *queuing time* of a message m_{ab}. The queuing time corresponds to the completeion time of a process, except that it does not include the transmission time of the message itself. As the message cannot be preempted during the transmission, queuing implies the time that it needs to wait before starting the transmission. Therefore the idea is to find the worst case queuing time due to other messages having equal or higher priorities.

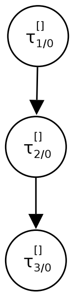
The analysis is extended by adding extra conditions to the reduction rules *Task-Interference* and modifying the contribution coming from instances arriving when $p > 0$. Another adjustment is done by introducing the maximum blocking time from *lower priority* messages, which is bounded by the maximum transmission time of all lower priority messages allocated to the same communication channel as m_{ab}

$$B_{ab} = \max_{\forall i, \forall m_{ij} \in lp_i(\tau_{ab})} C_{ij}^{m} \tag{3.23}$$

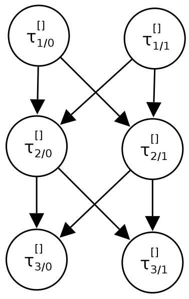
This maximum blocking time is used when finding the worst case queuing time, $q_{abc}(m_{ab})$, which is similar to equation (3.18), where the blocking time was ignored. In the rest of the algorithm the messages are handled as they were processes. We apply this approach direcly as defined by Redell.

3.2 Allowing Several Predecessors

The original analysis only supports one predecessor for each process, and in our case it will cause problems when using replication as fault tolerance technique. The reason being that when a process is replicated, its successors will have several predecessors. An example can be seen in Figure 3.5, where τ_2 and τ_3 will each get two predecessors when replication is added to all processes.



(a) The original process graph.



(b) The corresponding fault-tolerant process graph where all processes are protected with replication and $\kappa = 1$.

Figure 3.5: Example illustrating that replication will create several predecessors to processes, τ_2 and τ_3

Instead of focusing on the consequences of replication, we consider the problem more generally. We will therefore modify the algorithm, such that several predecessors are allowed, and thereby also including the special case of replication. In each of the following subsections we will start by describing the modifications strictly neccessary to allow several predecessors, and then try to reduce any pessimism introduced by the modifications.

We use $pred(\tau_{ab})$ to denote the set of immediate predecessors to process τ_{ab}, and $succ(\tau_{ab})$ as immediate successors to τ_{ab}. Starting from the very beginning, we need to consider how jitters and offsets need to be updated, when a process is preceded by several processes.

3.2.1 Offsets and Jitters

The offset represents the minimum delay for the arrival of process τ_{ab} due to the execution of preceding processes. It is given by the best case response time of the preceding process τ_{ap}, which implies that process τ_{ab} cannot start before its predecessor τ_{ap} has been executed.

When we have several predecessors, we can no longer compute the offset as defined in equation (3.1). However, we still want the offset to express the earliest possible arrival of τ_{ab}. And the offset then becomes the latest possible best case response time among all predecessors of process τ_{ab}

$$\Phi_{ab}{}^1 = \max_{\forall \tau_{ap} \in pred(\tau_{ab})} R^b_{ap} \tag{3.24}$$

This way of updating the offsets is very simple, and it works correctly when the predecessors run on different processing elements or when there is only one predecessor. However, if some of the predecessors are mapped on the same processing element, the value of the offset might be less correct. The reason being that the best case response time is found by adding the offset and the best case execution time C^b only. If some of the predecessors run on the same processing elements, the best case response times of the predecessors will be too optimistic (too low) since they will preempt each other and may force the analysis to consider some phasings that are not possible.

Nevertheless, this problem will not lead to incorrect worst case reponse times, because the preemption will be contained by jitters, as they depend on the worst case response time, R^w, that includes the interference from other processes. And since the phasing of processes and segments does depend on jitters, the computation of release times is still valid.

We can improve equation (3.24) and thereby fine tune the values of the offsets. It is possible, because we know that predecessors of τ_{ab} running on the same processing element may preempt each other. Therefore we propose a modification to the algorithm that can find more exact values of the offsets in order to speed up the convergence of the algorithm.

We start by grouping τ_{ab} predecessors into sets by their respective processing elements, as shown by equation (3.25) below

$$SPE(N_i, \tau_{ab}) = \{\tau_{ap} \in pred(\tau_{ab}) \mid \mathcal{M}(\tau_{ap}) = N_i\} \tag{3.25}$$

Each group is then considered to produce a total best case delay equal to the sum of all best cases execution times C^b, and an initial offset, equal to the least offset among all processes in the group. The sum of the initial offset and the total best case delay is treated as possible offset candidate, and the largest offset candidate is selected as the offset to process τ_{ab}

$$\Phi_{ab}{}^2 = \max_{\forall N_i \in \mathcal{N}} \left[\min_{\forall \tau_{ap} \in SPE(N_i, \tau_{ab})} (\Phi_{ap}) + \sum_{\forall \tau_{ap} \in SPE(N_i, \tau_{ab})} (C^b_{ap}) \right] \tag{3.26}$$

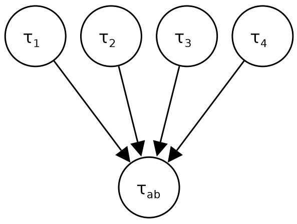
The offset found in equation (3.26) will be larger than (3.24) in some situations, but not all. To ensure that the offset is always on the safe side, we take the maximum of these two equations, such that the final version of the offset becomes

$$\Phi_{ab} = \max(\Phi_{ab}{}^1, \Phi_{ab}{}^2) \tag{3.27}$$

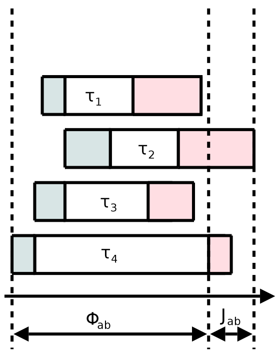
The jitters are found in a similar way to offsets. A jitter is used to express the maximum delay, a process τ_{ab} can expire after it has arrived at time Φ_{ab}. This maximum delay is therefore found as the difference between the earliest possible arrival of the process τ_{ab} and the latest time for release, i.e. when all predecessors are completed. If we look at the predecessors, we can notice, that the worst case response time of each predecessor contains interference from other processes, including other predecessors of τ_{ab}. Hence the predecessor having largest response time can be used to find the jitter of process τ_{ab} as follows:

$$J_{ab} = \max_{\forall \tau_{ap} \in pred(\tau_{ab})} R^w_{ap} - \max_{\forall \tau_{ap} \in pred(\tau_{ab})} R^b_{ap} \tag{3.28}$$

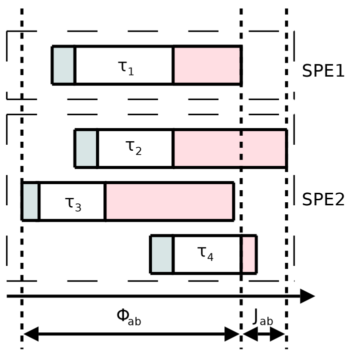
It is shown in Figure 3.6 how the offset and the jitter are found for τ_{ab}. The predecessors of τ_{ab} are $pred(\tau_{ab})$=$\{\tau_1,\ \tau_2,\ \tau_3,\ \tau_4\}$. The white boxes represent the best case response time, the light-blue boxes are offsets, and the pink boxes add the part from the worst case response time.



(a) A process graph, where process τ_{ab} has four predecessors.



(b) All processes run on different processing elements.



(c) Process τ_1 runs on N_1, all other processes run on N_2.

Figure 3.6: Illustrating the calculation of jitter and offset of process τ_{ab}.

In the first case, Figure 3.6(b), all predecessors are mapped to different processing elements, and the offset is found according to equation (3.24), i.e. the maximum best case response time among all predecessors, which is τ_4 in the example above. In Figure 3.6(c) processes τ_2, τ_3 and τ_4 run on the same processing element, and the offset is computed as the sum of their best case execution times plus the minimal offset among them.

After each iteration of the algorithm all local response times are re-computed. The jitters, the offsets and the global response times are updated for each process based on the newly found numbers. The offsets and the jitters of process τ_{ab} are computed with values retrieved from its predecessors. When considering just a single predecessor, updating of the processes is done recursively in depth-first way. In the cases with several predecessors, the changes must be propagated in a breadth-first way, because all predecessors of τ_{ab} must be updated before updating process τ_{ab} itself.

3.2.2 H-sections and H-segments

The next part of our modifications concerns changes in the definitions of H-segments and H-sections. Recall, that all processes of a segment will belong to the busy period of the process under analysis. In order to satisfy this property with several predecessors, we must derive a new definition of H-segments and H-sections.

The original requirement that all processes in a segment must be in $hp_i(\tau_{ab})$ is unchanged, but we need to redefine the condition whether two process from $hp_i(\tau_{ab})$ are in the same segment. Two processes having higher priority than τ_{ab} belong to the same segment, only if they have the same set of nearest ancestors that are not in $hp_i(\tau_{ab})$. When working on segments, this set of ancestors of process τ_{ik} is formally defined as:

$$\begin{aligned} SP^{seg}(\tau_{ab}, \tau_{ik}) \quad = \quad & \{\tau_{if} \in \Gamma_i \mid \notin hp_i(\tau_{ab}) \land \tau_{if} < \tau_{ik} \land \\ & (\neg\exists \tau_{il} \notin hp_i(\tau_{ab}) \mid \tau_{if} < \tau_{il} \land \tau_{il} < \tau_{ik})\} \end{aligned} \tag{3.29}$$

This requirement is also used for building H-sections, and two processes from $hp_i(\tau_{ab})$ are in the same H-section if they have identical ancestor sets, which are defined below:

$$\begin{aligned} SP(\tau_{ab}, \tau_{ik}) \quad = \quad & \{\tau_{if} \in lp_i(\tau_{ab}) \mid (\tau_{if} < \tau_{ik} \land \\ & (\neg\exists \tau_{il} \in lp_i(\tau_{ab}) \mid \tau_{if} < \tau_{il} \land \tau_{il} < \tau_{ik}))\} \end{aligned} \tag{3.30}$$

All *hp*-processes in Figure 3.7 have different SP-sets, as they have different ancestors $SP^{seg}(\tau_{ab}, \tau_4)=\{\tau_2\}$, $SP^{seg}(\tau_{ab}, \tau_5)=\{\tau_3\}$ $SP^{seg}(\tau_{ab}, \tau_6)=\{\tau_2,\tau_3\}$.

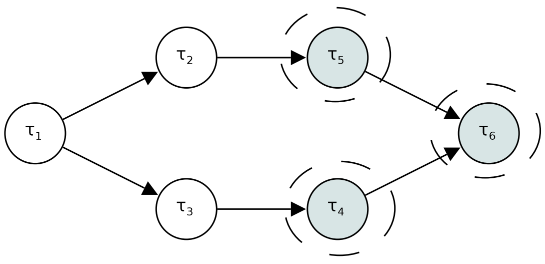


Figure 3.7: Since τ_4, τ_5 and τ_6 each have different SP sets, they are in different segments

Now, we can derive the updated definitions of H-segments

$$H_{ij}^{seg}(\tau_{ab}) = \{\tau_{ik} \in hp_i(\tau_{ab}) \mid SP^{seg}(\tau_{ab}, \tau_{ij}) = SP^{seg}(\tau_{ab}, \tau_{ik})\} \tag{3.31}$$

and H-sections

$$H_{ij}(\tau_{ab}) = \{\tau_{ik} \in hp_i(\tau_{ab}) \mid SP(\tau_{ab}, \tau_{ij}) = SP(\tau_{ab}, \tau_{ik})\} \tag{3.32}$$

The new definitions for H-sections and H-segments are backward compatible with the original ones, given by equations (3.9) and (3.8). The reason is because intermediate processes in $lp(\tau_{ab})$ are still not allowed. A process is said to precede an H-segment if it precedes all processes in the segment, $\tau_{ab} < H_{ij}^{seg}(\tau_{ab})$. As each process can have more than one predecessor, the set of immediate predecessors to an H-segment $pred(H_{ab}^{seg}(\tau_{ij}))$ has to be updated to reflect this:

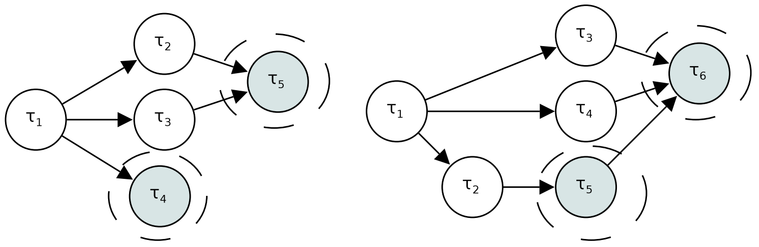
$$\begin{aligned} pred(H_{ij}^{seg}(\tau_{ab})) \quad = \quad & \{\tau_{ip} \in \Gamma_i | \tau_{ip} < H_{ij}^{seg}(\tau_{ab}) \wedge \\ & \big(\exists \tau_{ik} \in H_{ij}^{seg}(\tau_{ab}) | \tau_{ip} \in pred(\tau_{ik})\big)\} \end{aligned} \tag{3.33}$$

Similarly, the definition of $succ(H_{ij}^{seg}(\tau_{ab}))$ is updated in order to reflect that $pred(\tau_{ik})$ is not longer a single process, but a set of processes instead:

$$\begin{aligned} succ(H_{ij}^{seg}(\tau_{ab})) \quad = \quad & \{\tau_{ik} \in \Gamma_i | \tau_{ik} \notin H_{ij}^{seg}(\tau_{ab}) \wedge \\ & \big(\exists \tau_{il} \in pred(\tau_{ik}) | \tau_{il} \in H_{ij}^{seg}(\tau_{ab})\big)\} \end{aligned} \tag{3.34}$$

In the original version, an H-segment was blocking, if its predecessor (it could only have one) was in $lp(\tau_{ab})$. With more than one predecessor this no longer applies directly, and the question is whether a segment is blocking, when all its predecessors are in $lp(\tau_{ab})$. Looking at Figure 3.8(a), it can be seen that τ_4 and τ_5 defines two different segments, each consisting only of the process itself. It is also obvious that they are blocking, since τ_4 will always execute before τ_2 and τ_3, but τ_2 and τ_3 must always execute before τ_5. As a result τ_4 and τ_5 can not be in the same busy period and they will be blocking.

Considering Figure 3.8(b), processes τ_5 and τ_6 are non-blocking, since there might be an execution path as follows: τ_1, τ_3, τ_4, τ_2, τ_5, τ_6. In that case, τ_5 and τ_6 will be in the same busy period of τ_{ab}, and they are no longer mutually blocking. We therefore notice, that all predecessors $pred(H_{ij}^{seg}(\tau_{ab}))$ must be in $lp(\tau_{ab})$ for an H-segment to be blocking.



(a) Processes τ_4 and τ_5 each defines two segment which are blocking

(b) The segment defined by process τ_6 is not blocking

Figure 3.8: Example of Blocking and Non-blocking Segments

However, in some cases this assumption is not strong enough, as shown in the example in Figure 3.9. Neither segment H_4^{seg} or H_5^{seg} is blocking, even all predecessors are in $lp_i(\tau_{ab})$. They can both be in τ_{ab} busy period when the execution flow is τ_1, τ_3, τ_2 and so on.

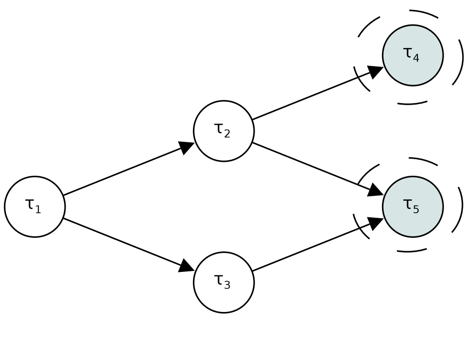


Figure 3.9: Example where segments H_4^{seg} and H_5^{seg} are not blocking according to equation (3.35).

Therefore a stronger condition for defining whether an H-segment is blocking or not is needed. Instead we say that not only the predecessors must be in

$lp_i(\tau_{ab})$, but also all successors to the predecessors that are not in the same H-section. More formally, the following must be satisfied for a given H-segment to be blocking:

$$\begin{aligned} & pred(H_{ij}^{seg}(\tau_{ab})) \subset lp_i(\tau_{ab}) \wedge \\ & \big(\forall \tau_{ik} \in pred(H_{ij}^{seg}(\tau_{ab})) \mid \\ & \big(\forall \tau_{il} \in succ(\tau_{ik}) \mid \tau_{il} \in H_{ij}^{seg}(\tau_{ab}) \vee \tau_{il} \in lp_i(\tau_{ab})\big)\big) \end{aligned} \tag{3.35}$$

The new definition will ensure that any two blocking segments cannot contribute to the same busy period of τ_{ab}. An informal proof follows. A segment is ready to run only when all its predecessors have finished executing. If there are no other *hp*-processes, which are not in the segment, following the predecessors, then the segment must be the only segment ready to run when all the predecessors have finished. Since all predecessors are in *lp*, there can not be any other segment ready to run at the same time.

Another definition that must be refined when considering several predecessors, is when a segment precedes a process. Redell defines this, as when the process does not belong to the segment and is preceded by $pred(H_{ij}^{seg}(\tau_{ab}))$. It will not hold in the case with several predecessors, which is shown in Figure 3.10.

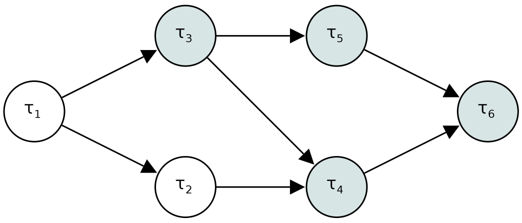


Figure 3.10: Example illustrating that a given process (τ_4) does not necessarily precede a segment ($H_{\tau_3}^{seg} = \{\tau_3, \tau_5\}$) when it is preceded by the predecessor of the segment (τ_1)

According to our definitions, there are two segments, $H_{\tau_3}^{seg} = \{\tau_3, \tau_5\}$ and $H_{\tau_4}^{seg} = \{\tau_4, \tau_6\}$. It is clear that even though τ_4 is preceded by the predecessor of the segment $H_{\tau_3}^{seg}$, τ_1, it is not proceeded by τ_5. Thereby it is not preceded by the segment as it is not preceded by all elements in the segment. Therefore we change the definition to be more concise

$$H_{ij}^{seg}(\tau_{ab}) < \tau_{ik} \text{ if and only if}$$

$$\big(\tau_{ik} \notin H_{ij}^{seg}(\tau_{ab})\big) \wedge \big(\forall \tau_{il} \in H_{ij}^{seg}(\tau_{ab}) | \tau_{il} < \tau_{ik}\big) \tag{3.36}$$

This definition is used only in some of the reduction rules of *TaskInterference*, which were mentioned in Section 3.1.4. The reduction rules are used to eliminate any processes that, even though they were part of a branch, could be eliminated anyway. When introducing several predecessors, we must consider these reduction rules and make sure that no processes are wrongly eliminated. Looking through each of these reductions rules, the only thing that must be changed concerns the elimination of processes that cannot occur in the same busy period because of the order they will be executed. This particular reduction is defined as part of rules 1a, 4b and 5b. We keep the following part from reduction rules

$$H_{aj}^{seg}(\tau_{ab}) < \tau_{ab} \tag{3.37}$$

but change

$$H_{ab}(\tau_{ab}) \neq H_{aj}(\tau_{ab}) \tag{3.38}$$

such that we are still sure that τ_{aj} will not interfere with τ_{ab} even with several predecessors. This is for sure fulfilled when there is a low-priority process between $H_{ab}(\tau_{ab})$ and τ_{aj}. In that case, the low-priority process will always ensure that both processes cannot occur for a given job in the same busy period. Therefore we change (3.38) to the following:

$$\exists \tau_{il} \in \Gamma_i \mid \tau_{il} \in lp_i(\tau_{ab}) \land H_{ik}^{seg}(\tau_{ab}) < \tau_{il} \land \tau_{il} < \tau_{ij} \tag{3.39}$$

Now, we can continue explaining the changes to be applied in *BranchInterference* that completes our modifications in the original algorithm to enable support for multiple predecessors.

3.2.3 Computing Transaction Interference

We will now consider the transaction interference computed by function *BranchInterference*, which was introduced in Section 3.1.4. When allowing several predecessors the function needs to be refined accordingly. In order to clarify the reasons for the changes, we will explain how the original version works and identify some possible pitfalls.

The function is a recursive function that traverses the process graph and finds the maximum contribution. The transaction is divided into *branches*, which are sub-parts of the transaction graph. Each branch starts at a process $\tau_{iB} \notin hp_i(\tau_{ab})$, and includes all processes preceded by τ_{iB}, which is called the *branch root process.* A branch may also contain other branches, also called *sub branches, SB.*

If τ_{iB} is a predecessor of an H-segment, then processes in $succ(H_{im}^{seg}(\tau_{ab}))$ will be the roots of the sub branches. Also those successors of τ_{iB} that are not in $H_{im}^{seg}(\tau_{ab})$ will start new sub branches. In Figure 3.11 processes τ_1, τ_4 and τ_5 define branches, with τ_4 and τ_5 being sub branches of the branch with root at τ_1.

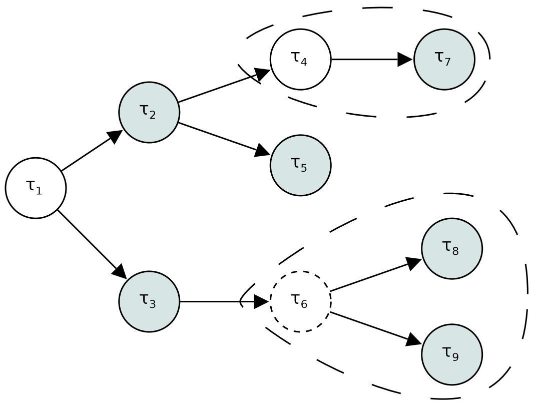


Figure 3.11: Branches in a Transaction

BranchInterference goes through all branches and finds the maximum non-blocking interference, *branchI*, achieved from each branch, if no blocking segment is allowed to execute. It also finds the maximum interference increase, *branchDelta*, if the branch is allowed to contribute with one blocking H-segment. The pseudocode of the original algorithm can be found in [33], Section 3.

A branch will contribute with *section interference*, *sectionI*, which is the total interference of all processes in section $H_{im}(\tau_{ab})$ preceded by branch root τ_{iB}

$$sectionI = \sum_{\tau_{ij} \in H_{im}(\tau_{ab})} TaskInterference(\ldots, \tau_{ij}, \ldots) \tag{3.40}$$

Please note that some parameters to *TaskInterference* in equation (3.40) were left out for the sake of simplicity.

In Figure 3.11 the branch starting at τ_1 may produce section interference with processes τ_2, τ_3, τ_5, τ_8 and τ_9. However, the interference of each particular process is computed according to the reduction rules in *TaskInterference*, and therefore some processes could be excluded from the section interference. Similarly, *sectionI* for the branch at τ_4 includes process τ_7, and for the branch at τ_6 is the sum of τ_8 and τ_9.

Besides the section interference, a branch can contribute with interference from its sub branches, *subBranchesI*, which is found as the sum of their individual branch interferences

$$subBranchesI = \sum_{\tau_{ix} \in SB} branchI(\tau_{iB}) \tag{3.41}$$

And the branch interference of the branch being analysed is found to be

$$branchI(\tau_{iB}) = \max(sectionI, subBranchesI) \tag{3.42}$$

because the processes in H^{seg}_{im} are in precedence conflict with other H-sections in the sub branches.

If we apply this approach directly on transactions with multiple predecessors, the algorithm may produce very pessimistic results. The pessimism is largely caused by the fact that some processes would be counted several times when finding the section interference.

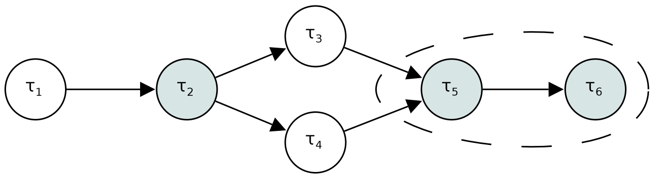


Figure 3.12: Pessimism in *BranchInterference*

Now we will consider the example in Figure 3.12. According to equations (3.40) - (3.42), the branch interference for the branch starting at τ_1, can either be the section interference of $H_{ab}(\tau_2)$ or the interference of its sub branches. By the original definition, there are two sub branches starting at τ_3 and τ_4, respectively. Therefore, the interference from the sub branches is $branchI(\tau_4) + branchI(\tau_4)$. As a result, processes τ_5 and τ_6 are added twice, once for each sub branch, resulting in a total interference that is too large and pessimistic. The reason is that it was originally assumed, that none of the sub branches would overlap. Therefore, in order to reduce the pessimism we propose following improvements to function *BranchInterference.*

Our solution to the problem is not to sum the process interferences directly, but group the processes that might contribute into sets, which we call *interference sets*. It means that we change equation (3.40) to a set

$$\begin{aligned} \mathit{sectionIS} &= \{\tau_{ij} \in AM(\tau_{iB}) \mid \\ &\quad TaskInterference(\ldots, \tau_{ij}, \ldots) > 0\} \end{aligned} \tag{3.43}$$

where AM is the set containing all processes from all sections in the branch that have τ_{iB} as immediate predecessor

$$AM(\tau_{iB}) = \{\tau_{ij} \in H_{im}(\tau_{ab}) \mid \tau_{iB} \in \mathit{pred}(H_{im}(\tau_{ab}))\} \tag{3.44}$$

The contribution of an interference set is computed as

$$\mathbb{C}(IS) = \sum_{\tau_{ij} \in IS} C_{ij} \tag{3.45}$$

which produces the same value as in the original version of $\mathit{sectionIS}$. Finally, the branch interference is updated to reflect the changes

$$\mathit{branchIS}(\tau_{iB}) = \begin{cases} \mathit{sectionIS}\ , \\ \qquad \texttt{if}\ \mathbb{C}(\mathit{sectionIS}) \geq \mathbb{C}(\mathit{subBranchesIS}) \\ \mathit{subBranchesIS}\ , \\ \qquad \texttt{if}\ \mathbb{C}(\mathit{sectionIS}) < \mathbb{C}(\mathit{subBranchesIS}) \end{cases} \tag{3.46}$$

and the total sub branch interference is now the union of all sub branches interference sets

$$\mathit{subBranchesIS} = \bigcup_{\tau_{iB} \in SB} \mathit{branchIS}(\tau_{iB}) \tag{3.47}$$

In equation (3.46) the branch interference is now found as the interference set that either represents the largest contribution among alls sub branches or the section interference of the current branch. The result now does not include duplicated process interference, as with the simple addition, and hereby the pessimism of *BranchInterference* is reduced as desired.

Yet another effect of introducing multiple predecessors is that now H-segments and H-sections may have immediate successors that are in $hp_i(\tau_{ab})$ because of different SP-sets. Therefore a sub branch root processes might also be in $hp_i(\tau_{ab})$ and therefore has to be included when computing section interference. If the branch root τ_{iB} precedes several segments, then none of them can be blocking according to (3.35) and the branch interference is found by (3.46). If the segment is blocking, the maximum non-blocking interference is $\mathbb{C}(\mathit{subBranchesIS})$, while the maximum interference increase is obtained as $\mathbb{C}(\mathit{sectionIS}) - \mathbb{C}(\mathit{subBranchesIS})$, that is in the same way as in the original algorithm.

We have included the pseudocode of the modified function *BranchInterference* in Appendix C. The parameters used in the pseudocode are written according to the original pseudocode, which is given in [34].

3.3 Conditional Analysis

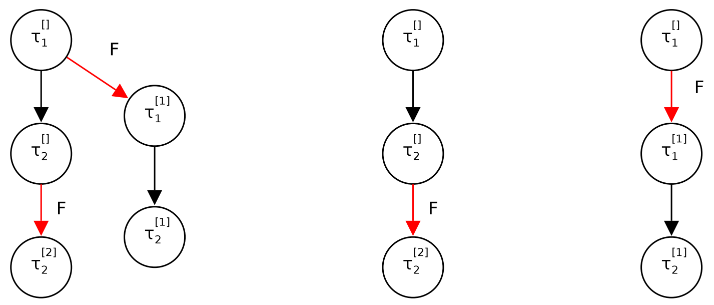
Now we will extend the algorithm further to provide conditional scheduling. As previously noted, this thesis addresses fault-tolerant applications that can be modelled as conditional process graphs, where conditions on edges serve as fault guards. There are several ways of computing process response times on such process graphs, depending on the requirements to the results. In the following we will describe some of them as described in [31], which can be directly combined with the response time analysis algorithm used in the thesis.

The first approach is called *ignoring conditions (IC)*. It is a very simple and naive solution, because the conditions are ignored completely. It means that the application graph is treated as a large unconditional graph. In this case, all processes are allowed to interfere with each other, even those that belong to mutually exclusive fault scenarios. This is why the results obtained by this simplified approximation will be very pessimistic. However, the IC-method might be used in some situations, when the precision is less important. An example could be if the designer only needs a fast, even if imprecise, estimate of the worst-case response times.

The second method is called *brute force (BF)* approach. The response times are computed by decomposing the original fault-tolerant process graph into the set of corresponding non-conditional graphs, each representing a unique fault scenario. The response time is therefore found for each scenario separately, making the results much less pessimistic. However, this method might be very time-consuming, as it requires a lot of computations to be repeated for each scenario.

The third approach, called *condition separation (CS)*, uses the fact that some processes are mutually exclusive because they are in different fault scenarios. Thereby we avoid the expected computational overhead by splitting into each of the scenarios as described by BF. Simultaneously, we should get more precise results than the IC-method. The remaining part of this section is dedicated to our proposal to modifications to WCDOPS+ algorithm, that are necessary to adopt it to CS-based response time analysis.

An example in Figure 3.13 shows why the pessimism of IC in some cases are not acceptable and CS or BF must be used instead. Notice that the response time found using IC for process τ_2 in Figure 3.13(a) is 5, thereby not meeting the deadline of 3. But as both faults cannot occur at the same time, it is shown in Figures 3.13(b) and 3.13(c) that the actual worst case response time for process τ_2 is 3.



(a) A Fault-Tolerant Process Graph. If ignoring conditions, then the worst case response time for τ_2 is 5 and the deadline is violated.

(b) The situation where process τ_1 fails. Worst case response time for for τ_2 is 3

(c) The situation where process τ_2 fails. Worst case response time for for τ_2 is 3

Figure 3.13: An example illustrating that Ignoring Conditions might falsely report violated deadlines. $P_1 = 2$, $P_2 = 1$ and $\kappa = 1$. All execution times are 1 and deadline for process τ_2 is 3.

The evaluations of the proposed methods can be found in Chapter 7, where the three approaches are compared according to computation time and the precision of the results.

3.3.1 Limitations of the Algorithm

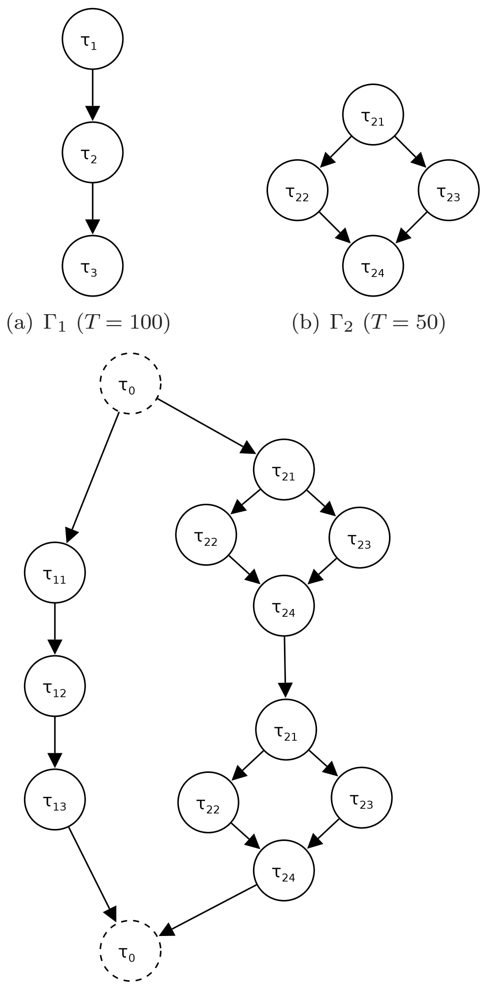
Before we start explaining how the *Condition Seperation* (CS) method can be used, it is important to clarify the assumptions that have been made when applying this approach.

The first problem will occur if we allow deadlines to be larger than periods. According to our fault model, the number of faults κ is the maximum number of faults during one period. If the execution of a process is postponed over several periods, the interferences from other instances become unpredictable. The response time analysis is based on the assumption that the transaction being executed is kept unchanged for every period. If two consecutive instances experience different faults, the transactions will be different. This will lead to an undetermined interference from instances that might be pending from previous event arrivals. For this reason it has been assumed that the deadlines of the processes can not be larger than the period the transaction.

Another limitation concerns the number of independent transactions. Assume that the system consists of more than one transaction. First we consider the situation where all process graphs have identical periods and are phased such that all arrivals of events occur simultaneously. In that case, we must consider all possible combinations of faults across all transactions. If we assume that κ is the total number of faults for all transactions, the number of scenarios to be considered is proportional to κ and the total number of processes. But if κ defines the number of faults in each transaction, the problem grows significantly. If we instead assume that each transaction defines its own period or the arrivals of the events do not occur simultaneously, the transactions will be shifted in some way. It causes a problem similar the one described above, because different instances of the same transaction might have different interferences to the other transactions. As a result the problem grows even more, and the previously described solution will not be feasible in terms of computation time.

Another point to be made, is that if the application model contains several independent transactions with different periods, there are several ways to interpret the meaning of tolerating κ faults. The simple solution would be to let κ define the maximal number of faults for each job of a transaction. Another solution is to let the fault model define κ for each transaction.

However, in some cases the designer can achieve fairly accurate results by merging all transaction into one large hyper-transaction and carry out RTA on it as it was a single transaction. An example is given in Figure 3.14. We assume that the designer have combined all transactions into a single hyper-transaction before using our analysis. The number of faults given in the fault model, will therefore be the total number of faults in the combined hyper-transaction. It should be acknowledged that this simplification does not take such things as event arrivals, synchronisation and the software architecture in general into account.



(a) Γ_1 ($T = 100$) (b) Γ_2 ($T = 50$)

(c) Γ_c ($T = 100$)

Figure 3.14: An Example of Merging Two Transactions (Γ_1, Γ_2) Into One Hypertransaction (Γ_c)

To summarise our limitations, the response time analysis presented in this thesis only works with applications containing a single transaction. Additionally, deadlines of processes are not allowed to be larger than the period of the transaction.

3.3.2 Reducing Interference

In this section we will describe how we modify the response time analysis algorithm to handle fault conditions. Since we use only two methods to provide fault tolerance, replication and reexecution, we have to analyse the consequences of each of the methods.

Considering replication no modifications are required as explained in the following. Recall that the definition of replication is that all replicas must be run in order to tolerate the appropriate number of faults. As we always assume the worst case situation, we must also always assume the worst case for the replicas. When considering the scheduling, this is the case when all replicas except one fails. In that case, the only to finish will be one with the longest response time. Thereby the execution of the sucessors to the replicas are delayed as much as possible. As a result, all replicas are considered independent processes that must be scheduled as any other processes. The general extension with several predecessors explained in Section 3.2, is therefore fully applicable in this situation.

If we use reexecution to protect a process against transient faults, the situation becomes slightly different. In contrast to replication, a process is only reexecuted, when a fault has happened. This information can be used to reduce the interference between those instances of processes that are mutually exclusive.

As described in the earlier sections, the algorithm finds the worst-case response time for each process. This is done by generating phasings of other processes in the transaction such that completion time of the process being analysed is delayed the most. A process can be delayed due to preemption by high priority processes, but some of the high priority processes cannot affect the execution of the process under analysis because of the precedence conflicts. Such conflicts are resolved by the reduction rules, and the consideration of mutually exclusive processes will be similar to that.

Two processes are said to be in *condition conflict*, if they belong to mutually exclusive parts of the fault tolerant process graph and therefore their executions will never overlap. By different scenarios we mean the conditional branches in the graphs, as shown in Figure 3.15.

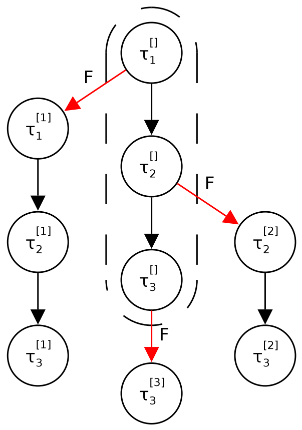


Figure 3.15: Condition Conflicts Example. Processes $\tau_1^{[1]}$, $\tau_2^{[2]}$ and $\tau_3^{[3]}$ are in different scenarios and may not interfere.

The figure demonstrates a transaction with three processes, all being protected with reexecution and $\kappa = 1$. If one of the processes fails, it will be reexecuted, which is captured by the conditional edge marked with F. The execution of this new branch created by the fault is incompatible with all other possible scenarios in the non-faulty branch. More details on how we define and find all scenarios are given in Chapter 4.

In this particular example κ is 1, and thus only one of three F-edges can be chosen during the transaction period, which means that there are four possible scenarios contained by the graph (including the one, when no faults occur). This reveals the meaning of condition conflicts - the processes are allowed to interfere within the same scenario only. For instance, if process τ_1 fails, then its reexecution will produce a new scenario with $\tau_1^{[1]}$ and $\tau_2^{[1]}$. These processes will contribute to the execution of other processes, nor they will be affected by it. The same is applicable for processes $\tau_2^{[2]}$, $\tau_3^{[2]}$ and $\tau_3^{[3]}$.

The solution is now to add an additional reduction rule to *TaskInterference* to resolve conditional conflicts, which can be stated as

$$SL(\tau_{ab}) \cap SL(\tau_{aj}) = \emptyset \tag{3.48}$$

where the set $SL(\tau_{ab})$ represents all scenarios, process τ_{ab} participates in. When this is satisfied, τ_{ab} and τ_{aj} will not interfere with each other. However, applying this reduction in *TaskInterference* is not very efficient, because a lot of work has to be done prior to *TaskInterference*. We can reduce the amount of time spent on the computations by inserting this reduction rule at outer stages of the response time analysis. Equation (3.22) corresponds to the outmost loop of the RTA algorithm used to find the worst-case response time of process τ_{ab}, when the busy period is started by τ_{ac}. If we include the reduction rule as a part of the loop, we can prevent the algorithm from wasting its time on the combinations of τ_{ab} and τ_{ac}, which are mutually exclusive. This is expressed as:

$$R^w_{ab} = \max_{\{\forall \tau_{ac} \in XP_a(\tau_{ab}) | SL(\tau_{ab}) \cap SL(\tau_{ac}) \neq \emptyset\}} \left[\max_{p_{ab}=p^{seg}_{0,abc}(\tau_{ab}) \ldots p^{seg}_{L,abc}(\tau_{ab})} R^w_{abc}(p_{ab}) \right] \tag{3.49}$$

Recalling that XP_{ac} contains all processes, that come first in their respective segments, we can improve the situation further by using (3.48) when building H-segments and sections. This will bring down the number of processes in segments and increase the speed of the algorithm further.

3.4 Pessimism and Performance

In the previous section we presented our approach of including condition separation in WCDOPS+. Processes that belong to different fault scenarios are mutually exclusive and thereby may not interfere. Nevertheless, in some cases this method is not capable of eliminating as much interference as when all scenarios are analysed separately. For instance, the reduction rule will be less effective if a transaction contains parallel chains of processes. If the chains are joined in another process, the reexecution may produce many scenarios that include all processes in the chains. This situation will be explained by the following example, and the corresponding process graph is shown in Figure 3.16.

The process graph shows a transaction where only process τ_3 is protected by reexecution. In principle the other processes may be protected as well, but in order to keep things simple they are not protected. Assume that all processes are mapped on the same processing element. Let processes τ_3 and τ_5 have higher priorities than process τ_2. The reexecution of τ_3 will produce new fault scenario starting at $\tau_3^{[3]}$. Process τ_2 will be included in this scenario, because $\tau_6^{[3]}$ joins the corresponding branches. As a result the algorithm is not able to eliminate the condition conflict between $\tau_5^{[\,]}$ and $\tau_3^{[3]}$ and both processes will influcence the response times of $\tau_2^{[\,]}$. Hence the response time of τ_2 becomes very pessimistic due to preemption by $\tau_3^{[\,]}$, $\tau_3^{[3]}$, $\tau_5^{[\,]}$ and $\tau_5^{[3]}$ even though some of them are mutually exclusive. Of course, if we apply brute force analysis, this problem never occurs. We can therefore expect that the results obtained by the condition separation approach will have longer response times compared to the brute force method.

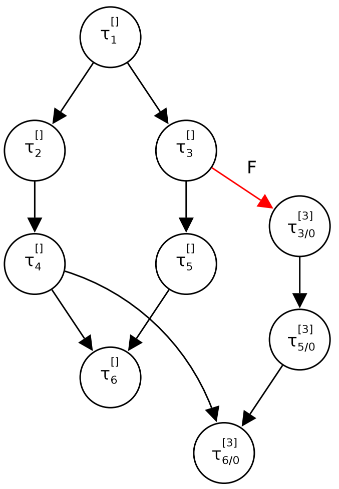


Figure 3.16: An Example Illustrating Potential Pessimism of the Condition Separation Approach. Processes $\tau_3^{[3]}$ and $\tau_5^{[\,]}$ will both interfere with process $\tau_2^{[\,]}$ even though they are mutually exclusive.

In addition we would like to discuss the execution time of the algorithm when it used on the fault-tolerant process graphs. As it was described in the beginning of this chapter, WCDOPS+ is able to compute upper bounds on the response times even if the deadlines are larger than periods. For reasons discussed in subsection 3.3.1, we only consider the case when the deadline is smaller than the period. In practice, this is modelled by letting the period be much larger than the deadline such that response times do not extend the period even when faults happen. This might have a negative impact on the execution time, because the algorithm will compute all possible phasing of all processes. As we only consider the case where the deadline is less than the period, we know beforehand that the phasing belonging to other periods of the transaction is not feasible. The algorithm unfortunately needs to calculate these phasings before it is able to realise that they are not able to interfere. We have not tried to evaluate how significant the problem is, nor optimised this particular matter. The reason is that it will not generally be the case that the period is smaller than the deadline and that a transaction cannot be interfered from other, differently phased, transactions.

We have now completed the theoretical foundation of the response time analysis used in this thesis. In Chapter 6 we present major details on the implementation of WCDOPS+ and the integration with the rest of our optimisation framework. The evaluation results are presented in Chapter 7.

Chapter 4

Fault-Tolerant Process Graphs

Process graphs are widely used to describe interacting processes, and there also exist several extensions that can model conditional flows in applications. In Chapter 2 we presented the concept of fault-tolerant process graphs, also abbreviated as FTPG. We have also described how we can analyse the response time of an application modelled by a fault-tolerant process graph. In this chapter we are going to give more details on how to build and manipulate such graphs.

As mentioned above, there exist several approaches to work with FTPGs [15, 24]. In [23] it is described how an FTPG with messages can be built. The proposed method only addresses reexecution of processes and not replication. In [32] it is shown how to build graphs that support both reexecution and replication but without messages. Another issue is that both sources describe how to build graphs but not how to update their structure during design transformations. Such transformations are needed when doing optimisation of policy assignment and mapping. Therefore, it will be much more efficient to do incremental updates to the graph instead of rebuilding the graph from scratch every time we change the configuration.

Due to these reasons we have proposed several graph transformation methods. As a result, we start by analysing and defining the elements and their properties in an FTPG in Section 4.1. After that, Section 4.2 explains the internal data structures used to maintain the graph, including two hash tables that speed up the operations on the graphs. In Section 4.3 we describe how to efficiently add and remove replication for a given process, and the operations concerning the reexecution are given in Section 4.4. For each of these two sections, we firstly present the solution without messages and then extend it to the situation with messages.

Notice that the algorithms proposed in this chapter are optimized for speed, while memory consumption has been given a lower priority. The reason being that most operations do not require much memory, whereas the traversing a graph can take a considerably large amount of time. Since we want to achieve fast performance when modifying the graph, we try to minimise the number of traversals by storing some information in the memory. Thus we assume that there is enough memory available.

The pseudocode listings of all algorithms proposed in this chapter can be found in Appendix C. The pseudocode listings are purposely rather detailed, since it should be possible to implement the algorithms without knowing too much about the theoretical details. Additionally, some listings contain references to the corresponding equations in this chapter or methods in the program source code.

4.1 Definitions

A fault-tolerant process graph, denoted $\mathcal{G}_i$, is a special version of a conditional process graph, since there can be *only one* boolean condition for set of transitions from a given process in the graph. Such a condition is called the *fault condition.* A fault condition becomes true if and only if the process associated with the condition fails. Likewise, *no fault* implies that the process did not fail.

Examples of FTPGs has been shown previously in Figures 2.8 and 2.9. The transition corresponding to when the fault condition is true, is marked with a red colour and possible tagged with an *F*. Black edges are taken when the condition is false. For a given process, we assume that there can only be one outgoing transition with a fault condition that is true.

A fault-tolerant process graph $\mathcal{G}_i$ is derived from the corresponding process graph Γ_i. The difference is that the processes in $\mathcal{G}_i$ are protected against faults. To protect a process against faults, we use the two fault-tolerance techniques described in Section 2.2: reexecution and replication. The set of processes in the $\mathcal{G}_i$ protected by reexecution is denoted $\mathcal{P}_x$. Correspondingly, the set of processes protected by replication is denoted $\mathcal{P}_r$.

If all processes are protected against faults, the process graph is said to be *fully protected*. For the operations described in the following, an FTPG does not have to be fully protected. In the real world there might be many reasons to have only partial protection of an application consisting of many processes. Therefore, we also allow to model applications, where only a subset of processes have to tolerate faults. On the other hand, it is required that any protected process in $\mathcal{G}_i$ must resist exactly κ faults.

In order to create and modify an FTPG, we must understand exactly how to distinguish the elements (also called *vertices*) in such a graph. First of all, a process τ_{ab} might be represented several times in the graph, for instance when we model reexecution. Looking at Figure 2.8 we can observe that process τ_4 occurs five times in the graph. What separates these five elements, are the conditions that model the faults happened prior to process τ_4 and in the process itself. These faults can be found by traversing the graph in Figure 2.8 upwards starting from each instance[1] of τ_4, as shown in Table 4.1.

Instance of τ_4	Faults happened	Processes executed prior to τ_4
τ_4	τ_2	τ_1, τ_2, τ_3, $\tau_{2/2}$
τ_4		τ_1, τ_2, τ_3
$\tau_{4/2}$	τ_4	τ_1, τ_2, τ_3, $\tau_{4/2}$
τ_4	τ_3	τ_1, τ_2, τ_3, $\tau_{3/2}$
τ_4	τ_1	τ_1, $\tau_{1/2}$, τ_2, τ_3

Table 4.1: Representation of τ_4 in Figure 2.8

The table shows for each of the five instances, which processes have executed previously. It is noticeable that the number of occurred faults is not enough to identify the different situations. For a given number of faults, the number of processes executed previously is fixed, but the combinations of elements are different. Since each process might have a different execution time, the influence from previous processes might be different. As consequence we would need to capture this difference by introducing a slack in order to hide, which process failed previously. Given that the goal is to avoid any slack, this approach is not feasible. Instead we use a list of faults that already have occured, as an unique property of each process. The list of previous faults, called *fault list*, is a multiset of the processes that have failed prior to that particular element in the graph. If a process has not failed nor does it precede the given process, it will not be included in the fault list. In case a process has failed more than once, it will be included in the list several times corresponding to the number of times it has failed previously.

[1]Notice that *instance* here means an occurrence of a process in the graph

However, in some cases the fault list would not be enough to uniquely describe a process either. Looking at Figure 2.9 it is obvious that replication causes the same process with identical fault lists to be represented several times in the graph. To capture this we introduce another property called *replica number*. Any replicas of a given process with a given fault list, are assigned consecutive numbers starting from 1 in order to uniquely identify these elements. As a result, each vertex in an FTPG is described by the properties shown in Table 4.2.

Property	Notation
Original process	τ_{ab}
Fault list	f
Replica number	r

Table 4.2: Properties of a Process in an FTPG

We will therefore discard the notation used in Section 2.2. Instead, each vertex of an FTPG will be denoted in the following way: $\tau^{f}_{ab/r}$. Here ab identifies the original process, f denotes the fault list and r is the replica number. We apply the same notation to messages, such that a message m_i, is now denoted $m^{f}_{ab/r}$. Notice that even though a message cannot be replicated, we still use the term replica number. As it will be described in Section 4.3, additional messages might be created when a process is replicated. We will, similarly to processes, assign increasing consecutive numbers to these messages such that the replica number of a given message corresponds to the replica number of the sender of the message.

As an example of this notation, please again consider Table 4.1. The five occurrences of process τ_4 will be now labelled as follows: $\tau^{[2]}_{4/0}$, $\tau^{[\,]}_{4/0}$, $\tau^{[4]}_{4/0}$, $\tau^{[3]}_{4/0}$, and $\tau^{[1]}_{4/0}$.

We denote the set containing all occurrences of a given process τ_{ab} in $\mathcal{G}_a$ as $RS(\tau_{ab})$:

$$RS(\tau_{ab}) = \{\tau^{f}_{ac/r} \in \mathcal{G} | ab = ac\} \tag{4.1}$$

For the sake of simplicity the replica number can be left out, such that $\tau^{f}_{ab/0}$ instead is written as τ^{f}_{ab}. In that case it is implied that $r = 0$. We will use the term *original process* for a process, $\tau^{f}_{ab/r}$, when $f = \emptyset$ and $r = 0$ that is the process existing in the initial process graph. The same applies for messages, in that case called an *original message*.

The *fault list* is a multiset of processes, such that a given process might exist several times in a given fault list. We define the function m to indicate how many occurrences of τ_{ab} there are in the fault list f:

$$m(\tau_{ab}, f) = |\{\tau_{ac} \in f | ab = ac\}| \tag{4.2}$$

Generally, a number of properties exists for a given FTPG. First, a fault list of a given process must always be a superset of all fault lists of all predecessors in the graph:

$$\forall \tau^{f_b}_{ab/r_b}, \tau^{f_c}_{ac/r_c} \in \mathcal{G}_a | \left(\tau^{f_b}_{ab/r_b} < \tau^{f_c}_{ac/r_c} \Rightarrow f_b \subset f_c \right) \tag{4.3}$$

If a process is replicated, then other processes in the fault-tolerant graph cannot have this process in their fault lists:

$$\forall \tau_{ab} \in \mathcal{P}_r \mid \left(\neg \exists \tau^{f_c}_{ac/r_c} \in \mathcal{G}_a | \tau_{ab} \in f_c \right) \tag{4.4}$$

Consequently, if a process is reexecuted, then there cannot be any occurrences of that process with a replica number different from zero:

$$\forall \tau_{ab} \in \mathcal{P}_x \mid \left(\neg \exists \tau^{f_c}_{ac/r_c} \in \mathcal{G}_a | r_c \neq 0 \right) \tag{4.5}$$

4.2 Data Structures

The internal representation of an FTPG is very simple. All vertices are doubly-linked, meaning that all processes know their immediate predecessors and successors. It is important to note that messages are also modelled as vertices in the graph. Hence, if a data dependency between processes τ_x and τ_y exists, such that τ_x sends message m_i to τ_y then m_i is modelled as a vertex between τ_x and τ_y.

There are several reasons for letting messages be vertices in the graph instead of being a property of a dependency edge. First of all it resembles the idea of a dependency: a process cannot start before the previous element, a process or message, has finished. Secondly, in the response time analysis, a message is modelled as a non-preemptive process. Therefore it makes sense to represent the message the same way as a process. Additionally, in our model the CAN-bus is used, so the messages are broadcasted and might have several recipients. In that case, the same message would need to be a property of several arcs. If the message is connected to all recipients, the model is much simpler. It implies that a vertex representing a message can only have one predecessor.

However, the doubly-linked structures are not very efficient in situations, when we need to search for elements. In order to improve the performance of operations on an FTPG, we will use two lookup tables to minimise the need for traversing the whole process graph. Notice, that the tables will not store any information that cannot be found by traversing the FTPG using the relations between the vertices. The lookup is done by using hash code of the keys, and therefore assumed to perform in constant time[2].

The first table is called LFTA. It provides lookup of processes in the graph via a key consisting of process τ_{ab} and a fault list f. The returned value will contain a process with a given fault list including all replicas of the very process. In Appendix E.2 an example of an FTPG is given in Figure E.7. The corresponding contents of LFTA are given in Table E.1.

The other table, referred to as LPFL, is used to locate all instances of a process τ_{ab} that have the given number of faults in their fault list. In other words, we can locate those instances of τ_{ab} that would be executed, if the application had experienced the given number of faults. The replicas are not stored in the table, and only processes with replica number $r = 0$ can be located via LPFL. The reason being that once $\tau^{f}_{ab/0}$ has been found, table LFTA can be used to obtain the corresponding replicas. Continuing the example from before, the contents of LPFL of Figure E.7 is given in Table E.2.

In the following sections we will clarify with examples, when and why these tables can be efficiently used.

4.3 Replication

4.3.1 Adding Replicas

In this section we will describe how to replicate process τ_{ab} in $\mathcal{G}_a$. Initially, τ_{ab} is not protected by any fault-tolerance at all, and after the replication it is fully protected against exactly κ faults. The steps of this procedure are described below.

We know that τ_{ab} might exist as a number of instances, $\tau^{f}_{ab/r}$, in the graph because its predecessors could be protected by reexecution. Each occurrence of τ_{ab} will have $r = 0$ and a fault list f describing the faults of its predecessors. Moreover, each occurrence of τ_{ab} must be protected against a number of faults that can be found as $\kappa - |f|$. In other words, the number of replicas for an occurrence of τ_{ab} will depend on the number of faults that already have been modelled prior to that occurrence.

[2]Assuming that the hash value can be computed in constant time.

In order to locate the processes that need to be replicated, table LPFL is used. Thereby we do not need to traverse the entire FTPG in order to locate all instances of the given process, but can locate these elements in constant time. We do a lookup $LPFL(\tau_{ab}, i), i = 0 \ldots \kappa$, and for all returned occurrences of process τ_{ab}we add the appropriate number of replicas. The pseudocode for locating the vertices is given in Algorithm 10 found in Appendix C, and it applies regardless of the presence of messages in the graph.

First we consider a $\mathcal{G}_i$ without messages and describe how to add replication to a given specific process, $\tau^f_{ab/0} \in \mathcal{G}_a$. This is done by inserting a new process, $\tau^f_{ab/r}$, into the graph connecting it to the same predecessors and successors as the original process being replicated. Notice that the replica number, r will be 1 for the first replica, 2 for the second and so forth. All replicas must have the same properties (execution time, period, mapping table etc) as the process being replicated. All predecessors of $\tau^f_{ab/0}$ must be connected to each $\tau^f_{ab/r}$

$$pred(\tau^f_{ab/0}) = pred(\tau^f_{ab/r}) \tag{4.6}$$

and likewise all replicas must be connected to the successors of $\tau^f_{ab/0}$, such that

$$succ(\tau^f_{ab/0}) = succ(\tau^f_{ab/r}) \tag{4.7}$$

An example of replicating a process is illustrated in Figure 4.1. The initial FTPG is given in Figure 4.1(a) and the result of replicating $\tau^{[\,]}_{4/0}$ is illustrated in Figure 4.1(a).

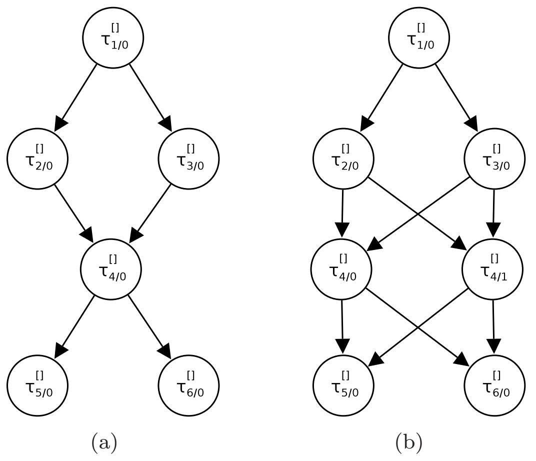


Figure 4.1: Replication of Process $\tau^{[\,]}_{4/0}$

If $\mathcal{G}_a$ contains messages, the approach is a little different. Recall that the communication bus used is CAN, and messages therefore are broadcasted. It means that any message sent to process $\tau^{f}_{ab/0}$, could also be received by any of its replica, $\tau^{f}_{ab/r}$. Therefore all incoming messages of the original process must be connected to all replicas, as defined by equation (4.6). On the other hand, a message can have only one sender, so any outgoing message $m_{ab(f)(0)}$ from $\tau^{f}_{ab/0}$ must be replicated and connected to the corresponding replicated process as its predecessor. More formally it can be defined as follows:

$$\forall m^{f}_{ac/0} \in succ(\tau^{f}_{ab/0}) \Rightarrow m^{f}_{ac/r} \in succ(\tau^{f}_{ab/r})$$

In Figure 4.2 it is illustrated how the replication is done, when messages are involved. Figure 4.2(a) is the original FTPG including two messages, and in Figure 4.2(b) process $\tau^{[\,]}_{4/0}$ has been replicated. Notice that $m_{1/0}$ is simply connected to the new replica, $\tau_{1/1}$, while $m_{2/0}$ is copied as $m_{2/1}$ and connected to the new replica, $\tau_{1/1}$.

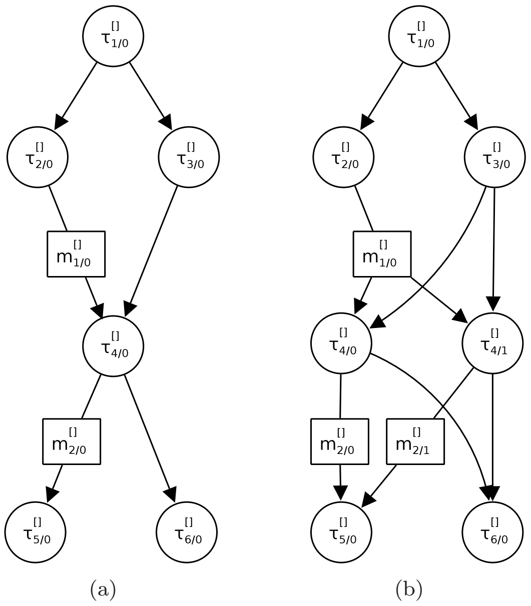


Figure 4.2: Replication of Process $\tau^{[\,]}_{4/0}$. Notice the difference between messages $m^{[\,]}_{1/0}$ and $m^{[\,]}_{2/0}$

An important detail is that a replicated process is not added to the fault list of its successors. The reason being even though a process is protected by replication, we do not know whether a fault will actually happen in the process or in any of its replicas. Therefore, we always model the worst case, implying that no faults have been tolerated. As a result, fault lists of the successors of the replicated process will always be same as of its predecessors.

4.3.2 Removing Replicas

In order to disable replication of process τ_{ab} we must locate all instances of τ_{ab} in the graph and remove all the replicas. More formally, the processes to be removed are defined as follows:

$$\left\{ \tau^{f}_{ac/r} \in \mathcal{G}_a \middle| \, (ab = ac) \wedge (r > 0) \right\}$$

We utilise the lookup tables in order to locate the vertices in $\mathcal{G}_a$ that must be removed. Using LPFL we locate all occurrences of τ_{ab}, $|f| < \kappa$. We do not consider $|f| = \kappa$ since we know that this will not have any replicas. For each of the found vertices, we find the corresponding list of processes in LFTA and remove all elements except the first one, which is the 0^{th} replica.

We return to process replication in Sections 5.1 and 5.2, where we discuss, how to select an appropriate priority and mapping for replicated processes and messages. Furthermore, Algorithms 11 and 14 in Appendix C list the pseudocode for adding and removing replicas of a given process.

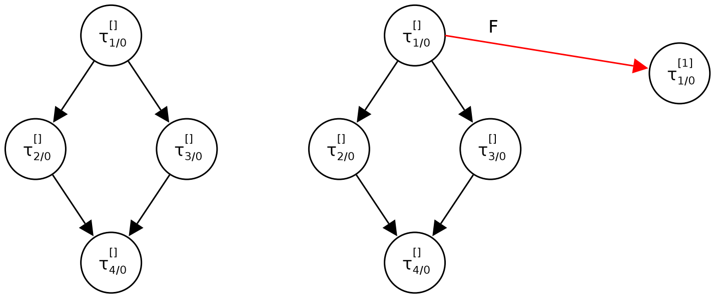
4.4 Reexecution

4.4.1 Adding Reexecuting

In this section we will describe how to add reexecution for process τ_{ab}, which is not protected by any fault tolerance initially. As with replication we recognise that τ_{ab} can exist as several vertices in the graph due to faults from other preceding processes. Notice that since we cannot combine reexecution and replication, then r must be 0 for all occurrences of τ_{ab} and f is unique for each of vertices. We also assume that after having added reexecution the given process must be fully protected against all κ possible faults.

We start by looking at the case, when there are no messages in the graph and the reexecution takes place on the same processing element. First we locate all vertices $\tau^{f}_{ab/r} \in \mathcal{G}_a$ that represent all occurrences of process τ_{ab}. Each of these will have predecessors that might have experienced $|f|$ faults, and for that reason each occurrence needs to be protected against the remaining number of faults $\kappa - |f|$, so the total number of tolerated faults becomes κ In the following we will describe how add a single reexecution, but this process is simply repeated, when adding more reexecutions,.

We model reexecution of process $\tau^{f}_{ab/r}$ by adding another process, $\tau^{f \uplus ab}_{ab/r}$ as a successor to $\tau^{f}_{ab/r}$. In the following we will call $\tau^{f}_{ab/r}$ *failing process* and $\tau^{f \uplus ab}_{ab/r}$ is called *reexecuting process*. The edge between $\tau^{f}_{ab/r}$ to $\tau^{f \uplus ab}_{ab/r}$ is guarded by a fault-condition as explained previously in subsection 4.3.1. This transition models the situation when $\tau^{f}_{ab/r}$ fails and for that case, no other transitions can be taken. If the process does not fail, then all other transitions are taken, but not the edge with the fault-condition. With $f \uplus ab$ in the faultlist of $\tau^{f \uplus ab}_{ab/r}$ we indicate that the process τ_{ab} is added to the fault list f since we know that this transition is taken if and only if $\tau^{f}_{ab/r}$ has failed. This part is illustrated in the example below. The graph in Figure 4.3(a) is the original process graph, and Figure 4.3(b) shows an itermediate graph after having created and connected the reexecuting process $\tau^{[1]}_{1/0}$.



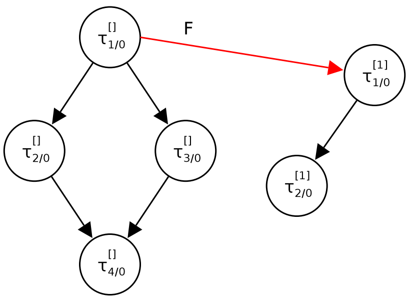
(a) Initial graph (b) Intermediate graph

Figure 4.3: Reexecution of Process τ_1 in Progress, Part I. Process $\tau^{[1]}_{1/0}$ models the reexecution of $\tau^{[]}_{1/0}$.

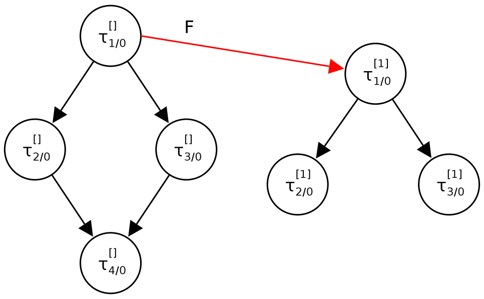
The next step is to create and connect the missing successors of $\tau^{f\uplus ab}_{ab/r}$. We cannot let the successors of $\tau^{f}_{ab/r}$ be the successors of $\tau^{f}_{ab/r}$ directly, as we did with replication. The reason being that they will have a fault list smaller than that of $\tau^{f\uplus ab}_{ab/r}$, which, according to the assumption in equation (4.3), does not make sense. Therefore we must, for each original successor of τ_{ab}, $\tau_{ac} \in succOrg(\tau_{ab})$, create a corresponding process $\tau^{f\uplus ab}_{ac}$ as a successor of $\tau^{f\uplus ab}_{ab/r}$. Notice that the new successors will have the same fault list as the reexecuting process, $\tau^{f\uplus ab}_{ab/r}$. This procedure continues recursively meaning that we need to copy the successors of successors and so forth. The process graph from the previous example can be seen in Figure 4.4, where the remaining processes are added.

In the following we will describe in detail how new process are created during the recursion mentioned above. We assume that we start in $\tau^{f}_{ab/r}$ and would like to create a successor of $\tau^{f\uplus ab}_{ab/r}$ corresponding to process τ_{ac}. We divide the problem into two different situations:

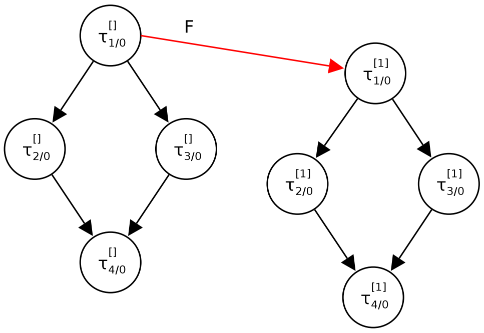
- Process τ_{ac} has one predecessor.
- Process τ_{ac} has more than one predecessor.



(a) Intermediate graph



(b) Intermediate graph



(c) Resulting graph

Figure 4.4: Reexecution of Process τ_1 In Progress, Part II. The missing processes are being added.

If process τ_{ac} has only one predecessor, we simply create a new process, $\tau^f_{ac/r}$, such that the fault list is equal to $\tau^f_{ab/r}$. Then process $\tau^f_{ab/r}$ is connected with $\tau^f_{ac/r}$ and the recursion continues for $\tau^f_{ac/r}$. This is illustrated in Figures 4.4(a) and 4.4(b), where successors $\tau^1_{2/0}$ and $\tau^1_{3/0}$ are added to the reexecuting process

$\tau^{1}_{1/0}$.

When process τ_{ac} has several predecessors, the situation is more complex. The predecessors of the new copy τ_{ac} must correspond to the predecessors of the original process τ_{ac}. Returning to the example from before, please consider Figure 4.4(b). We continue from $\tau^{[1]}_{3/0}$ and want to create τ_4 as a successor. Process τ_4 has two predecessors, τ_2 and τ_3, which implies that the new copy of process τ_4 must also have the corresponding τ_2 and τ_3 as its predecessors. Additionally, we know that the fault list for the new process, must contain the fault, τ_1, from $\tau^{[1]}_{2/0}$. Therefore we create $\tau^{[1]}_{4/0}$ and connect it to the only instances of processes τ_2 and τ_3 that have τ_1 in their fault lists, that is to say $\tau^{1}_{2/0}$ and $\tau^{1}_{3/0}$. The result is shown in Figure 4.4(c).

Things become even more difficult when different faults must be combined. More generally, we can consider $\tau^{f_b}_{ab/r_b}$ with three original predecessors, τ_{ax}, τ_{ay}, and τ_{az}. As each of the three predecessors might exist several times in $\mathcal{G}_a$ with different fault lists, we define all possible combinations, APC, of these as the Cartesian product of their occurrences:

$$APC = RS(\tau_{ax}) \times RS(\tau_{ay}) \times RS(\tau_{az})$$

Notice that an element, $apc_i \in APC$, is only valid if it fulfils all of the following permutation requirements:

- The combined fault list must be superset of f_b
- If the fault list of $\tau^{f_q}_{aq/r_q} \in apc_i$, contains τ_w, $\tau_w \in f_q$, then all other elements of apc_i that have τ_w as a predecessor must also have experienced this fault:

$$\Big\{\forall \tau^{f_q}_{aq/r_q} \in apc_i, \forall \tau^{f_r}_{ar/r_r} \in apc_i/\tau^{f_q}_{aq/r_q} \mid (\forall \tau_s \in f_q \\ \mid (\tau_s \in pl(\tau_{ar}) \Rightarrow (m(\tau_s, f_q) = m(\tau_s, f_r))))\Big\}$$

- The size of the combined fault list does not exceed κ

Please consider the situation shown in Figure 4.4(a). We have added $\tau^{[1]}_{2/0}$ and would like to create the copy of τ_4. Process τ_4 has predecessors τ_2 and τ_3, so we create the possible permutations:

$$\begin{aligned} APC &= RS(\tau_2) \times RS(\tau_3) \\ &= \{\tau^{[\,]}_{2/0}, \tau^{[1]}_{2/0}\} \times \{\tau^{[\,]}_{3/0}\} \\ &= \{\tau^{[\,]}_{2/0}, \tau^{[\,]}_{3/0}\}, \{\tau^{[1]}_{2/0}, \tau^{[\,]}_{3/0}\} \end{aligned}$$

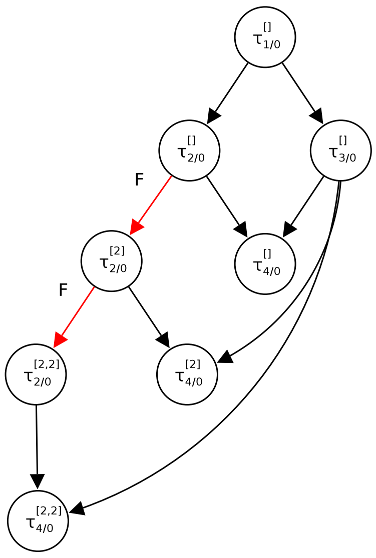
Combination $\left\{\tau_{2/0}^{[\,]}, \tau_{3/0}^{[\,]}\right\}$ is not valid because the combined fault list, $f = \emptyset$, is not a superset of the fault list of $\tau_{2/0}^{[1]}$, $f = [1]$. The second combination, $\left\{\tau_{2/0}^{[1]}, \tau_{3/0}^{[\,]}\right\}$, does not satisfy the third condition since τ_1 is in the fault list of $\tau_{2/0}^{[1]}$ then it must also be in the fault list of $\tau_{3/0}$. This is, however, not the case. We take the next step shown in Figure 4.4(b), where process $\tau_{3/0}^{[1]}$ has been added. Now we can generate more combinations:

$$\begin{aligned}
APC &= RS(\tau_2) \times RS(\tau_3) \\
&= \{\tau_{2/0}^{[\,]}, \tau_{2/}^{[1]}\} \times \{\tau_{3/0}^{[\,]}, \tau_{3/0}^{[1]}\} \\
&= \{\tau_{2/0}^{[\,]}, \tau_{3/0}^{[\,]}\}, \{\tau_{2/}^{[1]}, \tau_{3/0}^{[\,]}\}, \\
&\quad\ \{\tau_{2/}^{[0]}, \tau_{3/0}^{[1]}\}, \{\tau_{2/}^{[1]}, \tau_{3/0}^{[1]}\}
\end{aligned}$$

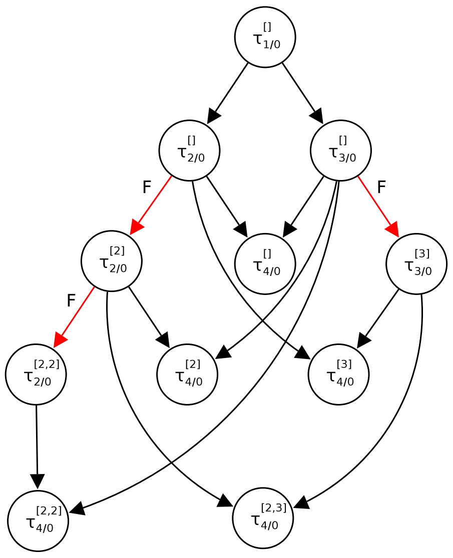
Checking the permutation requirements, we see that only $\left\{\tau_{2/}^{[1]}, \tau_{3/0}^{[1]}\right\}$ is valid. Therefore when process $\tau_{4/}^{[1]}$ is created, it should have $\tau_{2/}^{[1]}$ and $\tau_{3/0}^{[1]}$ as predecessors. The resulting graph can be seen in Figure 4.4(c).

Finally, we consider another example with $\kappa = 2$, shown in Figure 4.5. The graph already contains reexecution of process τ_2, and we want to add reexecution of process τ_3. When we add first reexecution of τ_3, we need to add two occurrences of τ_4, as show in Figure 4.5(c), because we also need to model two faults happening in τ_2 and τ_3. Then we add additional reexecution of τ_3 in order to model that all two faults take place in process τ_3.

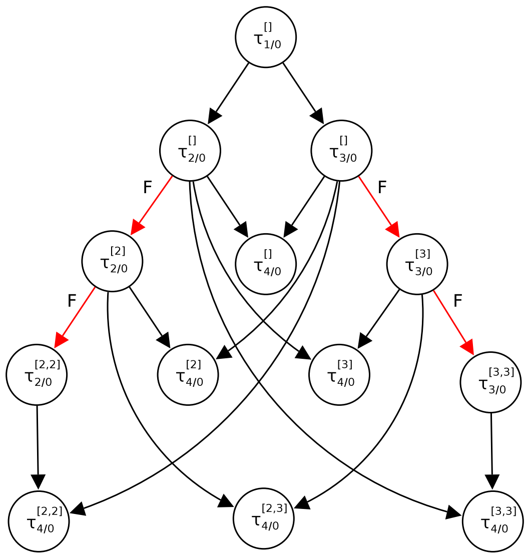
When having found one or more valid permutations, we do the following for each permutation: First it is checked, whether the process already exists. If it does, nothing is to be done, since we know that it must already be connected appropriately. If it does not exist, we create a copy and insert into the graph with the set of predecessors as the elements of the combination. For this new element, we apply this method recursively.



(a) Initial graph



(b) First reexecution of process τ_3



(c) Second reexecution of process τ_3

Figure 4.5: Adding Reexecution When $\kappa = 2$. The figure illustrates, how we add reexecutions of process τ_3, whereas τ_2 is already reexecuted.

4.4.2 Termination of Recursion

The recursion described in the previous section terminates when either of the following is fulfilled:

- It has reached an original process with no successors.
- There are no possible combinations.

The first reason is straightforward and will not be discussed further. In order to understand the second reason, please consider the examples shown in Figures 4.3 and 4.4. In the situation shown in 4.4(a) there were no feasible combinations for process τ_4, and only process $\tau^1_{3/0}$ was created. Therefore, the recursion stopped at $\tau^1_{2/0}$ and took next successor at $\tau^1_{3/0}$. Process $\tau^1_{4/0}$ was firstly created, when a valid combination of its predecessors became available that is after $\tau^1_{3/0}$ was created. We claim without proof that if there are missing permutations of predecessors for given process, then these permutations will always be created later in the recursion. It means that if a given permutation is not found, then it must surely be caused by missing processes that always will be created by another "branch" of the recursion. And subsequently we will be able to create the given permutation when all necessary processes have been created.

4.4.3 Combining with Replication

Since reexecution of a process involves copying of its successors, we need to consider situations, when some of them must be protected by replication or reexecution. Therefore we have to ensure that we have the right number of replicas for each replicated process that is preceded by the reexecuting process. Also, if some of the successors are reexecuted, then the number of reexecutions should be correct too.

When we add reexecution of a process, we do not copy any replicas while building sub trees of its successors. During the recursion, we check for each new process whether it needs any replicas or reexecutions. An example follows.

Assume that we are adding reexecution for process τ_{ab}. The corresponding reexecuting process $\tau^{ab}_{ab/r}$ has been created, and we are now copying the successors of τ_{ab} that will be preceded by $\tau^{ab}_{ab/r}$. If we meet a successor $\tau_{ac} \in \mathcal{P}_x$, the algorithm will add the remaining number of reexecutions of τ_{ac} before continuing with the remaining successors of τ_{ab}.

If process τ_{ac} belongs to $\mathcal{P}_r$, then we add it to the list of awaiting processes that need replication. Only when adding the reexecution of τ_{ab} is completely finished, the algorithm will create the required number of replicas to each process in the list of awaiting processes. At that point, when the recursion is completed, we are certain that all successors of $\tau^{f}_{ab/r}$ have been created. Therefore we can simply use the method explained in subsection 4.3.1 to add the remaining replicas.

4.4.4 Reexecution with Messages

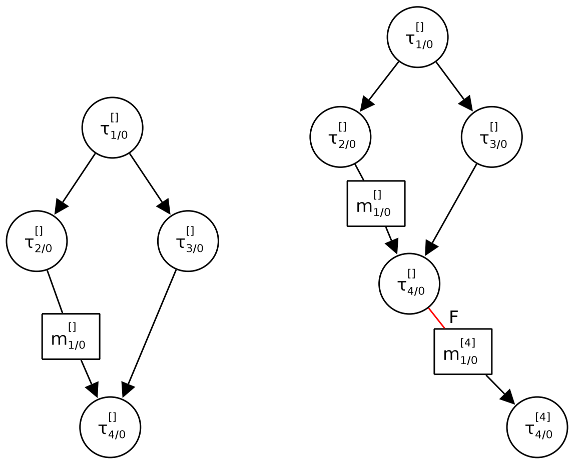
In the following we explain how we model the communications, when the process being reexecuted receives and sends messages. The first situation to be considered is, when a process is reexecuted on the same processing element. As defined by our hardware model, received messages are stored in the communication subsystem. It means that the reexecuting process can obtain the required messages directly from the communication subsystem. This property eliminates the need to resend the messages, when a process fails. So the incoming messages received by process being reexecuted will not be copied. On the other hand any messages that the original process will send should also be sent by any of its reexecutions, if they do not fails. Therefore the outgoing messages must be repeated for each reexecution, i.e. modelled as its successors.

When the reexecution of a process takes place on another processing element, the messages must be treated in a different manner. This way of reexecution is also called *passive replication.* Let process $\tau^{f}_{ab/r}$ be the failing process, $\tau^{f\uplus ab}_{ab/r}$ the reexecuting process and $\mathcal{M}(\tau^{f}_{ab/r}) \neq \mathcal{M}(\tau^{f\uplus ab}_{ab/r})$. We divide the problem into two different cases:

- One message.
- Several messages.

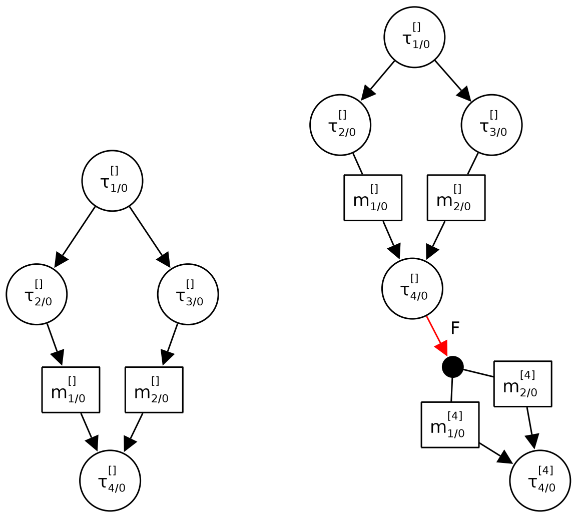
When there is a single message, we repeat that message between the failing process and the reexecuting process. The message is placed on the same communication bus as the original message. A situation with one message is illustrated in Figure 4.6.

The situation is somewhat different when there are several messages. We assumed previously that a process can have only one fault-condition. To avoid having several edges with a fault-condition, we introduce a *dummy-process* to fork the messages being repeated. An example with two messages is shown in Figure 4.7.



(a) Initial graph (b) Message is resent

Figure 4.6: Reexecution of process τ_4 with one incoming message that must be retransmitted, because the reexecution happens on another processing element..



(a) Initial graph (b) Dummy process is used

Figure 4.7: Reexecution of process τ_4 with two incoming messages that must be retransmitted, because the reexecution happens on another processing element.

The dummy process is represented as a black circle in the graph. Dummy processes have no execution time so they will be transparent to the response time analysis. The messages to be retransmitted are placed between a dummy process and the following reexecuting process.

4.4.5 Removing Rexecution

Removal of the reexecution of process τ_{ab} is done by several steps. First all processes in $\mathcal{G}_a$ where $|f| < \kappa$ are located via LPFL. From these we remove the reexecuting process and then recursively remove all of its successors from $\mathcal{G}_a$. At the same time we wipe out any dependencies and messages from the remaining part of the graph that may point to those successors. The pseudocode for removal of reexecutions is listed in Algorithm 12 in Appendix C.

4.5 Remapping

So far, we have discussed how replication and reexecution are applied to a fault tolerant process graph. Now we will discuss the effect on messages, when a process in the FTPG is remapped. Assume that a given message, $m_{aj}^{f_b}$, is sent from $\tau_{ab}^{f_b}$ to $\tau_{ac}^{f_c}$. Processes $\tau_{ab}^{f_b}$ and $\tau_{ac}^{f_c}$ are assumed to be running on different processing elements. If the mapping of either $\tau_{ab}^{f_b}$ or $\tau_{ac}^{f_c}$ is changed such that both will execute on the same processing element, there is no longer need to transmit message $m_{aj}^{f_b}$ over the communication bus.

As a result, we set the transmission time of the message to zero, if the sending and all receiving processes are on the same processing element. This can be formulated generally as follows, where C_i^{mob} and C_i^{mo} is the transmission time over the bus:

$$C_i^m = \begin{cases} C_i^m = 0, & \forall \tau_{as}^{f_s} \in succ(m_i) | \mathcal{M}(\tau_{as}^{f_s}) = \mathcal{M}(pred(m_i)) \\ \\ C_i^m = C_i^{mo}, & \exists \tau_{as}^{f_s} \in succ(m_i) | \mathcal{M}(\tau_{as}^{f_s}) \neq \mathcal{M}(pred(m_i)) \end{cases} \tag{4.8}$$

and likewise for the best case transmission time:

$$C_{m_i}^{mb} = \begin{cases} C_i^{mb} = 0, & \forall \tau_{as}^{f_s} \in succ(m_i) | \mathcal{M}(\tau_{as}^{f_s}) = \mathcal{M}(pred(m_i)) \\ \\ C_i^{mb} = C_i^{mob}, & \exists \tau_{as}^{f_s} \in succ(m_i) | \mathcal{M}(\tau_{as}^{f_s}) \neq \mathcal{M}(pred(m_i)) \end{cases} \tag{4.9}$$

4.6 Defining and Seperating Scenarios

The concept of *fault scenario* was shortly introduced in the preliminaries. In this section we will define fault scenarios in a formal way, and explain, how they are extracted from a fault-tolerant process graph. We will also give a few examples to illustrate the concept.

A fault scenario s_{is} is a specific trace or execution path through an FTPG for a certain combination of faults. Therefore, each scenario is uniquely defined by the faults experienced through the trace. This can also be formulated as the fault lists of its sink nodes. Each of these unique fault lists contains up to κ elements, hence a fault list with n elements always will be a subset of one or more fault lists with $n+1$ faults. An exception is a fault list with $n = \kappa$, where no more faults can happen. Also notice that since replication hides any fault occurrence from succeeding processes, only processes protected by reexecution can be in these fault lists.

The set of all possible fault scenarios for a given FTPG represents all possible combinations of faults experienced by processes protected by reexecution. It also includes combinations with less than κ faults. A given process in the FTPG $\tau^{f}_{ab/r}$ might participate in several scenarios. We define the function SL that returns the set of scenarios in which the given process arise:

$$SL(\tau^{f}_{ab/r}) = \left\{ \mathrm{s}_i \in \mathcal{S}_a | \tau^{f}_{ab/r} \in \mathrm{s}_i \right\} \tag{4.10}$$

In order to find all fault scenarios in an FTPG, we start by locating all unique fault lists. Each list is assigned a unique number, and we locate and mark all processes that have a fault list being subset of the given unique fault list. Each of these sets of processes will in turn represent a fault scenario. A pseudocode for this method, is given in Algorithm 2.

We have included two examples in the Appendix E.1.2 showing how fault scenarios are found. Example 1 in Figure E.1.1 is a diamond-shaped transaction with all processes protected by reexecution. It gives five scenarios as shown in the Figures E.2-E.3. Example 2 in Figure E.1.2 shows the same transaction, but now with two processes protected by replication instead of reexecution. The derived faults scenarios can be seen too.

Algorithm 2 Pseudocode for Splitting FTPG into Scenarios

```
Require: $\mathcal{G}_i$, the FTPG to be split
  setUniqueFL ⇐ The set of unique fault lists of the FTPG
  scenarioNumber ⇐ 1
  for all $uniqueFL \in setUniqueFL$ do
    S ⇐initialise new scenario
    Assign scenarioNumber to S
    for all $\forall\tau^{f}_{ab/r} \in \mathcal{G}_a$ do
      if $f \subset uniqueFL$ then
        add $\tau^{f}_{ab/r}$ to S
      end if
    end for
    scenarioNumber ⇐ scenarioNumber + 1
  end for
```

4.7 Counting Processes and Scenarios

In this section we would like to quantify, how many processes and scenarios an FTPG can contain. We consider process τ_{ab} in the computations below.

As explained previously, such a process might be presented several times in the graph. If we assume that process τ_{ab} is reexecuted, then each occurrence in of the process the FTPG will be characterised by a unique fault list. The problem is to find the number of different fault lists, in which process τ_{ab} can participate.

We know that the processes in the fault list must precede τ_{ab}, and we also know that process τ_{ab} itself can be includes in the list. Moreover, only processes protected by reexecution can be in the fault list. We start by defining the function $PX(\tau_{ab})$, which returns the processes in the graph that precede τ_{ab} and are protected by reexecution. These processes will therefore exactly define the possible elements in the fault list of τ_{ab}:

$$PX(\tau_{ab}) = \{\tau_{ac} \in \Gamma_a | \tau_{ac} \in \mathcal{P}_x \wedge (\tau_{ac} < \tau_{ab} \vee \tau_{ab} = \tau_{ac})\} \tag{4.11}$$

From the combinatorics [25] we know that the number of ways that n different elements can be combined in order to form a list of size k, is given by the following binomial coefficient:

$$\binom{n+k-1}{n} = \frac{(n+k-1)!}{k!((n+k-1)-k)!} = \frac{(n+k-1)!}{k!(n-1)!} \tag{4.12}$$

If we replace n with the size of $PX(\tau_{ab})$ it is possible to compute the number of different fault lists with exactly k faults for process τ_{ab}. As we must include all values of κ, we must also include those fault lists with size less than κ. Therefore, we sum equation (4.12) for $k = 0..\kappa$. As we know that process τ_{ab} can only exist once with the empty list, we end up with the following function $NO(\tau_{ab}, \kappa)$ that expresses the number of occurrences of process τ_{ab} in $\mathcal{G}_a$:

$$NO(\tau_{ab}, \kappa) = 1 + \sum_{i=1}^{\kappa} \frac{(|PX(\tau_{ab})| + i - 1)!}{i!(|PX(\tau_{ab})| - 1)!} \tag{4.13}$$

Likewise we can find how many scenarios there are in an FTPG since the possible elements are defined by the set of processes protected by reexecution:

$$|\mathcal{S}_a| = 1 + \sum_{i=1}^{\kappa} \frac{(|\mathcal{P}_x| + i - 1)!}{i!(|\mathcal{P}_x| - 1)!} \tag{4.14}$$

Here $\mathcal{P}_x$ corresponds to the set of processes in the corresponding process graph Γ_a than must be reexecuted.

In Figure 4.8 we have illustrated the number of scenarios for different numbers of κ and processes being reexecuted. It can be seen that the number of scenarios increase exponentially with the number of processes. When the number of faults increase, the number of scenarios increases even faster.

Figure 4.9 shows the number of processes in the fault-tolerant process graph versus the number of process in the original graph for different values of κ. The maximum number is obtained when the processes are connected one-by-one and form a chain. The minimum number is obtained from transactions where all processes are immediate successors of a root process. In order to show upper bounds all processes are protected with reexecution only. We notice that for just 50 processes and 4 faults, we get a huge number of processes in the fault-tolerant process graph: $(3.5 \cdot 10^6)$.

From Figure 4.8 and 4.9 we see that the number of processes increases much faster than the number of scenarios on the average. Any response time analysis should therefore preferably depend on the number of scenarios rather than the number of processes.

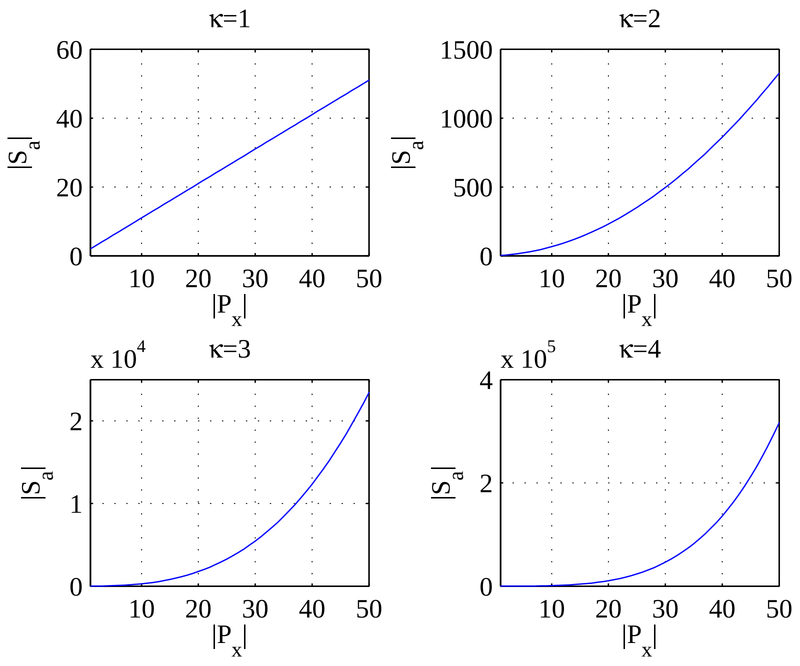


Figure 4.8: The number of scenarios is increasing exponentially for an increasing number of processes. When κ increases, the number of scenarios increases even faster.

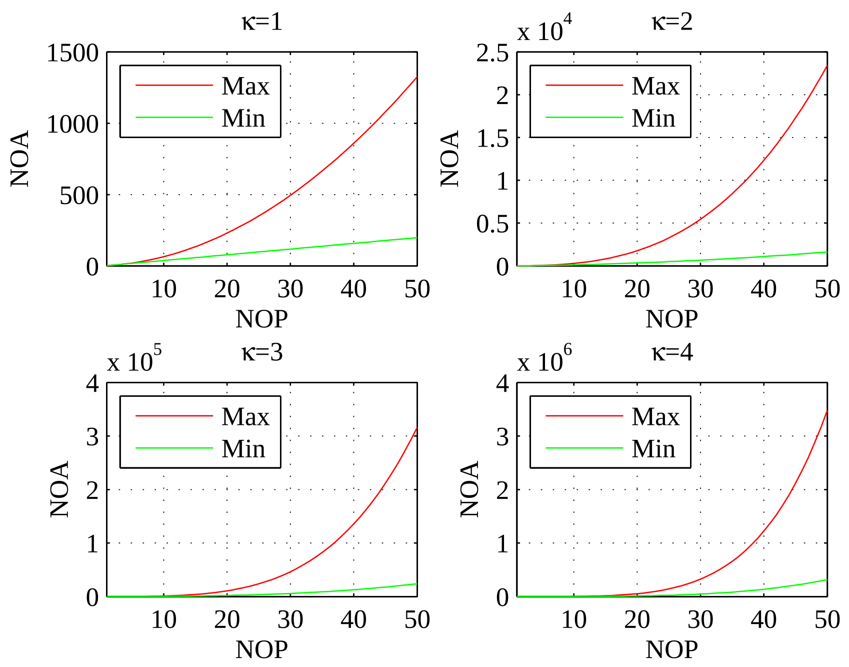


Figure 4.9: Number of processes in the fault-tolerant process graph (NOA) versus the number of processes in the original process graph (NOP) for different values of κ. The asymptotic best and worst cases have been illustrated. All processes are protected by reexecution. It can be seen that both cases are increasing exponentially.

In this chapter we have shown how replication and reexecution are applied to a process graph including messages. The algorithms also support configurations where only some of the processes are protected against faults. The operations described are optimised by using two hash tables to sort and group the vertices of the graph for faster lookups.

It would be relatively simple to extend the algorithms, such that different processes would be able to tolerate different numbers of faults. This would, among other things, require a discussion on how faults should be interpreted. If each process needs to tolerate a given number of faults, does this number include faults from preceding processes, or is it the maximum number of faults that the process can experience itself?

Chapter 5

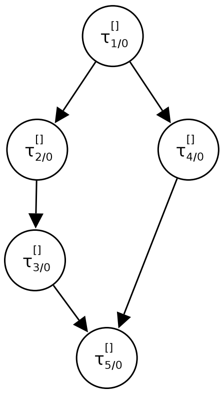
Fault-Tolerance Policy Assignment and Mapping

In the previous chapters we have explained how to do response time analysis (RTA) for event-driven fault-tolerant systems. Furthermore, it has been explained how to create and transform fault-tolerant process graphs that represent the application. This chapter will describe how we can help the designer in deciding the best combination of policy assignment and mappings to optimise the solution and satisfy the timing requirements.

In Figure 5.1 an example of an application is given. We would like to protect the application against one transitient fault and all processes have a deadline of 16. In Figure 5.2 all processes are protected with replication, but we can see that the deadlines are not met. Correspondingly, protecting all processes with reexecution in Figure 5.3 also violates the deadlines. Only by combing replication and reexection as shown in Figure 5.4 the deadlines are met. Therefore the designer needs a technique to automatically find such a schedulable solution.

Given an application, an architecture and a fault model, a designer could manually find a feasible solution that satisfies all the imposed requirements. Our schedulability analysis would then be used to check the timing properties of each design alternative derived manually. Unfortunately, the solution space is often very large, so such an entirely manual approach is not feasible in most situations.

Instead, we will propose an optimisation heuristic in order to automatically find the best possible solution within the solution space.

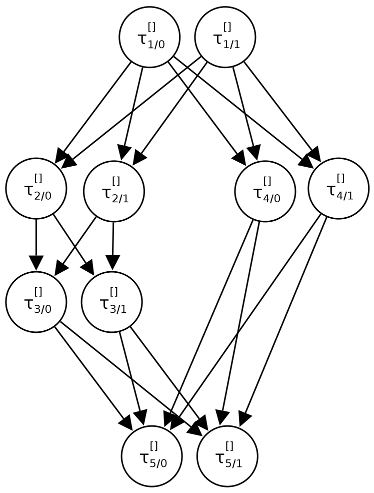


(a) Process graph

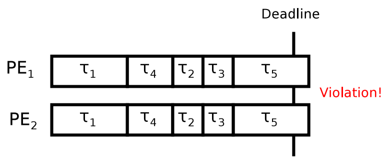
	P_i	N_1	N_2
τ_1	1	(5,5)	(5,5)
τ_2	1	(2,2)	(2,2)
τ_3	1	(2,2)	(2,2)
τ_4	2	(3,3)	(3,3)
τ_5	1	(5,5)	(5,5)

(b) Priorities and the mapping table for the processes

Figure 5.1: An example application



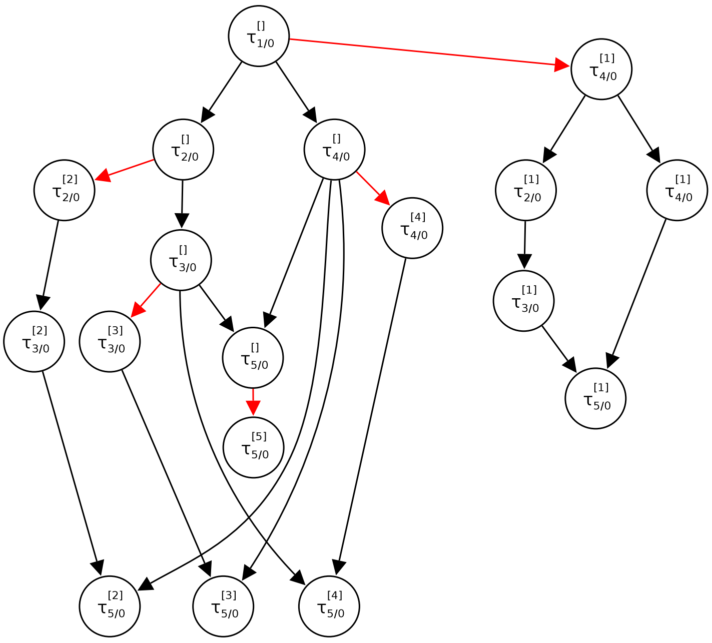
(a) All processes are protected by replication.



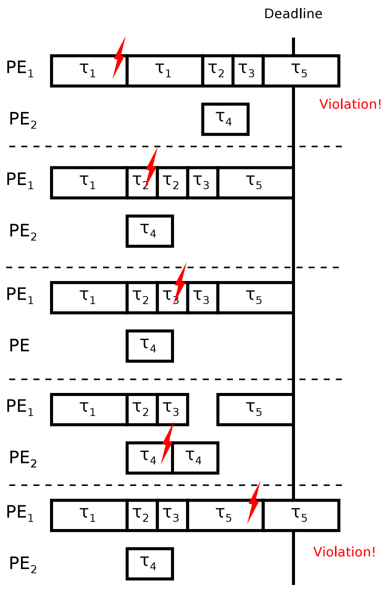
Deadline
PE_1
τ_1 τ_4 τ_2 τ_3 τ_5
Violation!
PE_2
τ_1 τ_4 τ_2 τ_3 τ_5

(b) Deadline is not met by process τ_5.

Figure 5.2: Illustrating that only using replication in the process graph in Figure 5.1 will violate the deadline.

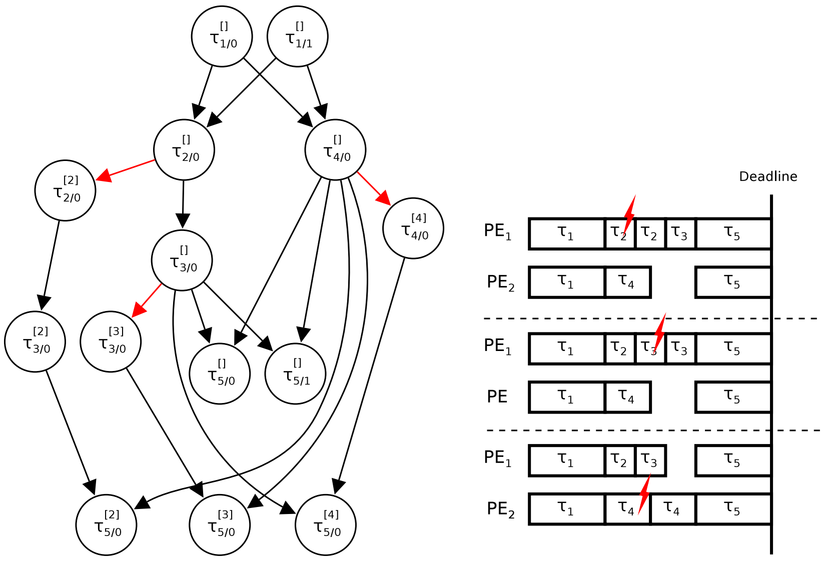


(a) All processes are protected by reexecution. Deadline is not met by neither $\tau_{5/0}^{[1]}$ nor $\tau_{5/0}^{[5]}$



(b) Deadline is not met when either process τ_1 or process τ_5 fail.

Figure 5.3: Illustrating that only using reexecution in the process graph in Figure 5.1 will violate the deadline.



(a) Processes τ_1 and τ_5 are protected by replication, while processes τ_2, τ_3, and τ_4 are protected with reexecution.

(b) The deadline is met in all fault scenarios.

Figure 5.4: Illustrating that when combining reexecution and replication for application in Figure 5.1 the deadline is met.

The optimisation problem that we would like to solve can be defined as follows:

- Which fault-tolerance technique to use for each process:

$$\tau_j \in \{\texttt{REPLICATION}, \texttt{REEXECUTION}\}$$

- The mapping of each process in the fault-tolerant process graph:

$$\mathcal{M}(\tau_{i/r}^{f})$$

The primary objective of the optimisation is to ensure that the system is schedulable for all possible scenarios. Once schedulable, the heuristics should minimise the response times using the given hardware resources.

In Sections 5.1 and 5.2 it will be discussed how priorities and mappings should be chosen initially, when processes are reexecuted or replicated. In Section 5.3 the optimisation problem itself will be defined. In the following subsections, we will discuss more technical matters, such as defining the neighbourhood to a given solution, the cost function to evaluate different solutions and also choose an appropriate heuristic. Finally, we will comment on the convergence of the heuristic in subsection 7.1.3.

5.1 Choosing Priorities

When creating replicas and reexecutions, the question is what priority they should be assigned. A straightforward solution would be to assign the same priority as the original processes. The second solution would to be to assign a different priority. This leads to a more fundamental discussion about what the purpose of priorities is. Recall that in the introduction, we stated that we would only be considering fixed priority scheduling. But this is only related to fact that the priority of each process does not change during execution. The question is whether it makes sense to change the priorities of the processes given by the designer.

The main purpose of the priorities is to tell the scheduler in which order the processes should be executed. In that way, the designer can control the scheduling and in what order processes finish execution, if it is not explicit defined by any dependencies. A more abstract interpretation of the priorities given by the designer could be *how important* the processes are. In that case, it would make sense to prioritise more important process higher than lower priority processes in the evaluation of a given configuration.

Considering the aspect of schedulability, the initial priority given by the designer no longer makes sense when fault-tolerance is applied to the system. The reason being the initial scheduling will be changed completely, and therefore it should be possible to change the priorities of the processes after having added fault-tolerance. This should apply to both replicated and reexecuted processes, and the same process with different fault lists. It should be possible, since the priorities will still be fixed during runtime even though the same original process might have different priorities during execution.

For the issue of importance of processes, the initial priority might still be used for this purpose even though the priorities are changed when applying fault tolerance. In that case, one must distinguish between the two types of priorities: a fixed *importance priority* for every process, τ_{ab}, and the *schedulability priority* of each process in the FTPG.

In [18] a specialised heuristic for selecting the optimal priority assignment for a distributed system is given. This article also explains that selecting a priority is not trivial.

To keep things simple, we have chosen to let the priorities always be fixed and not to integrate the ideas from [18]. As a result, the initial priorities given by the designer will be used as the priorities for any new processes introduced as a result of applying the fault tolerance.

5.2 Choosing Processing Elements for Replicas

When introducing replication, there are several approaches for deciding the initial mapping of the replicas. Even though the heuristic will optimise this choice later, a better initial choice will reduce the number of iterations required by the heuristics to converge. First of all, there might be limitations on the possible mappings defined in the mapping table. Among the allowed processing elements, there are several approaches for the mapping:

- A random processing element.
- The processing element on which the process has the lowest execution time.
- The processing element with the lowest utilisation, considering only elements which are not mutually exclusive to the replica.

Each approach has drawbacks. The first approach is completely naive, since it does not consider other processes and the utilisation of processing elements. The second approach picks the fastest processing element, but does not consider the fact that all other processes might already be assigned to this particular processing element. In that case, the response time will not reflect that the execution time is low. The latter approach uses a simple guess of picking the processing element on which it creates the least interference, but does not consider any dependencies. Being the most qualified guess, we will pick the third approach.

The utilisation of a processing element N_i is a measure of how much it is used by different processes. It can more formally be defined as the sum of the execution times of the processes currently mapped on the PE divided by the period of the processes. Since we only need the utilisation to compare between processing elements and the period is equal for all processes, we do not divide by the period. Notice, that only processes, which have at least one scenario common with τ_{ab}^{f}, will be able to influence the utilisation of N_i. The utilisation for N_i in the scenarios defined by $\tau^f_{ab/r}$ is therefore given by:

$$\rho(N_n, \tau^f_{ab/r}) = \sum_{\tau^{f_c}_{ac/r_c} \in SU(N_n, \tau^f_{ab/r})} C^w_{ac(f_c)(r_c)}$$

SU defines the set of processes that are not mutually exclusive to $\tau^f_{ab/r}$ and are mapped onto the given processing element N_i:

$$SU(N_n, \tau^f_{ab/r}) = \left\{ \tau^{f_c}_{ac/r_c} \in \mathcal{G}_a \mid \left(SL(\tau^f_{ab/r}) \cap SL(\tau^{f_c}_{ac/r_c}) \neq \emptyset \right) \wedge N_n = \mathcal{M}(\tau^{f_c}_{ac/r_c}) \right\}$$

So when creating a replica, $\tau^f_{ab/r}$, the initial mapping is decided as follows:

$$\mathcal{M}(\tau^f_{ab/r}) = \min_{N_i \in MA(\tau_{ab})} \rho(N_i, \tau^f_{ab/r})$$

When additional messages are introduced due to the replication of a process, then all these messages are assigned to the same communication bus as the original message. The reason is that we in the problem definition assumed that all messages are statically mapped.

5.3 Optimizing Policy Mapping

In this section, we will motivate and discuss how meta-heuristics can be used to find a policy assignment and mapping that leads to a schedulable, optimal solution. In subsection 5.3.1 we will explain the neighbourhood in terms of what moves are possible from a given solution. Implementing the objective as a cost function is explained and discussed in subsection 5.3.2, such that we will be able to compare different solutions. In subsection 5.3.3 the choice of initial solution is explained. Subsection 5.3.4 will be a discussion on possible different heuristics to use and details on the one chosen to be implemented. The convergence and results will finally be discussed in subsection 7.1.3.

5.3.1 Neighbourhood and Moves

In this section we will define the neighbourhood to a given solution. The neighbourhood is given by the set of design transformations that are possible. We also call such a design transformation a *move*. Based on the problem formulated in the previous section, we define two possible moves:

- **FaultToleranceMove**, change the fault tolerance technique for a given original process, τ_{ab}, to either replication or reexecution.
- **RemapMove**, change the mapping of a given process, $\tau^f_{ab/r}$.

A number of restrictions apply to the two moves. For the FaultToleranceMove, only processes, which can be mapped to at least κ different processing elements, can be replicated. Recall, that the number of possible mappings is defined by the mapping table given by the designer. The set of possible FaultToleranceMoves is therefore restricted to the changing all processes currently protected by replication to reexecution and changing processes currently under reexecution fulfilling the previous requirement, to replication.

For each process $\tau^f_{ab/r} \in \mathcal{G}_a$ we define a RemapMove for all possible mappings different from the current mapping. It requires that $\tau^f_{ab/r}$ can be mapped to more than a single processing element. Considering an example, then if $\mathcal{M}(\tau^f_{ab/r}) = N_n$ for the current solution, and $MA(\tau_{ab}) = \{N_n, N_o, N_p\}$, then it would be possible to remap $\tau^f_{ab/r}$ to N_o and N_p, respectively. An exception is when a process is replicated. In that case, any two replicas by definition cannot be mapped to the same processing element. Therefore a RemapMove cannot include processing elements that another replica is already assigned to. The processing elements that are allowed for $\tau^f_{ab/r}$ are defined by the following set:

$$\begin{aligned} PPE(\tau^f_{ab/r}) &= MA(\tau_{ab}) \setminus \{N_i \in MA(\tau_{ab}) | \\ &\qquad \Big(\neg\exists \tau^{f_c}_{ac/r_c} \in \mathcal{G}_a | ab = ac \wedge f = f_c \wedge \mathcal{M}(\tau^{f_c}_{ac/r_c}) = N_i\Big)\Big\} \end{aligned}$$

For each replica we create a move for each element in the set given above. It implies that if the number replicas including the process being replicated is equal to the number of possible mappings, then there are no possible moves.

Notice that the size of the neighbourhood depends very much on the problem and on the current solution. While the number of FaultToleranceMoves will depend on the number of original processes, the number of RemapMoves will be related to the number of elements in the FTPG and the contents of the mapping table.

5.3.2 Objective Function

In order to quantify how good a given solution is, we define an *objective function*, also called a *cost function*. Buttazzo defines in [5] a number of different possible cost functions such as *Maximum lateness* and *Maximum number of late tasks*. The problem with the cost functions in [5], especially when used for comparing two different solutions, is that either they do not include the deadline of the processes or they are a maximum over something. In the latter case, information about the processes not covered by the maximum is disregarded. As a result, the cost function does not fully represent, how good a solution is. Therefore we will use the degree of schedulability (DOS) as introduced in [31]:

$$DOS(\Gamma_a) = \begin{cases} c_1 = \sum_{i=1}^{n} \max(0, R_i^w - D_i) & \text{, if } c_1 > 0 \\ c_2 = \sum_{i=1}^{n} (R_i^w - D_i) & \text{, if } c_1 = 0 \end{cases} \tag{5.1}$$

This equation has two objectives. First, if some deadlines are not met (case c_1), then it will report the total lateness. So the first objective is to eliminate any violations of deadlines. Only when the first objective is fulfilled and all deadlines are met, then the response time of all processes are minimised. But the formula is defined for regular process graphs, and therefore cannot be applied directly to fault-tolerant process graphs. Whereas a given original process only exists once in a regular process graph, the same process might exist several times in the fault-tolerant process graph.

As a result, two different solutions might have different number of processes. An example is shown in Figures 5.5 and 5.6 with 12 processes and 9 processes, respectively. The problem is that the DOS for a graph is found by adding degrees of schedulability of each process in the graph. However, in our case the value of the cost function should not depend on the number of processes in a graph. So which processes in an FTPG should be included when calculating the degree of schedulability?

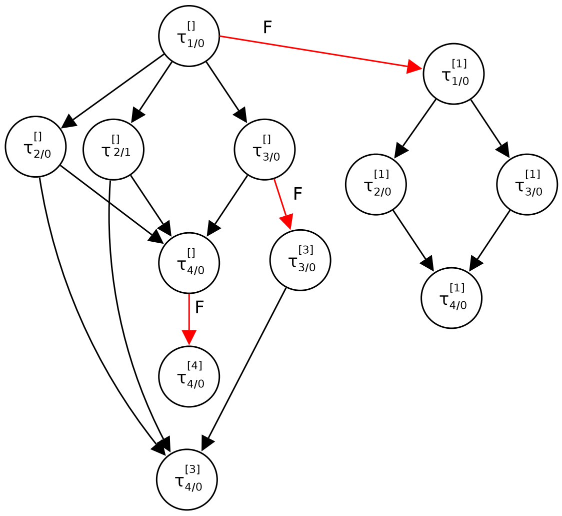


Figure 5.5: Process τ_2 is replicated, processes τ_1, τ_3 and τ_4 are reexecuted. There are a total of 12 processes and 4 scenarios.

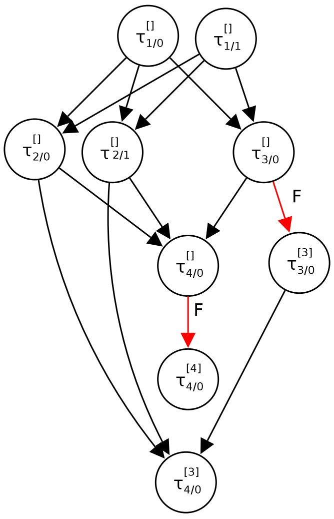


Figure 5.6: Processes τ_1, τ_2 are replicated, processes τ_3 and τ_4 are reexecuted. There are 9 processes and 3 scenarios.

A given process in the fault-tolerant process graph might have a different response time in each fault scenario. Considering equation (5.1) again, it does not allow a given process to have different response times and therefore cannot be used directly.

We will instead formulate a new equation based on (5.1), that accounts for the fault-tolerance. First we sum up the degree of schedulability as defined in (5.1) from each of the scenarios and divide by the number of scenarios:

$$DOS(\mathcal{G}_a) = \frac{\sum\limits_{\mathrm{s}_i \in \mathcal{S}_a} DOS(\mathrm{s}_i)}{|\mathcal{S}_a|}$$

Here $\mathcal{S}_a$ is the set of scenarios for the given solution. Notice that we divide by the number of scenarios, since two solutions might not have the same number of scenarios. An example is shown in Figures 5.5 and 5.6, where the number of scenarios is 4 and 3, respectively.

Some scenarios are more likely to occur than others. Since we assume that faults are independent, experiencing i faults are more likely than $i + 1$ faults. We therefore introduce a weight, w_i, to the different fault scenarios:

$$DOS(\mathcal{G}_a) = \frac{\sum\limits_{\mathrm{s}_i \in \mathcal{S}_a} w_i \cdot DOS(\mathrm{s}_i)}{|\mathcal{S}_a|}$$

Here the weight is given by

$$w_i = \frac{1}{1 + NF_{\mathrm{s}_i}} \tag{5.2}$$

NF_{s_i} defines the number of faults occurring in scenario s_i. The weights will ensure that scenarios with fewer faults have more influence on the DOS than scenarios with more faults. Even though we have decided the approach above, these weights should in principle be chosen by the designer, because he has more knowledge about the components of the system. Another possibility could be to include the probability of faults occurring on the different processing elements. Different processing elements will most likely have different probabilities of faults, because of the different quality of components, exposure to external radiation, size, and so forth. This could be combined with the response time of the processes, since a process that runs for a longer time, has a higher probability of failing than another running for a shorter period of time. Generally, the more concrete information is included in the model, the more accurate the estimate becomes. It should be emphasised, that our implementation is based on the simple model with the weight given by equation (5.2).

Now we need to define which elements should be included in the degree of schedulability as defined in equation (5.1). At first we recall that a process might exist in several instances in a given scenario. What is really interesting for each process is when we are completely sure that a process has successfully completed its execution. It includes any possible reexecutions or replicas if failures happen. Hence, if a process is reexecuted several times, it has only finished successfully when the last process in the chain of reexecutions has finished. Therefore only this process is included in the calculation of the degree of schedulability. For replication, we are only sure that the process has finished successfully when the slowest replica has finished. It is so, because we always assume the worst case which implies that the fastest replicas will fail. As a result, we take the process with the largest response time among the replicas. We therefore define the set of the processes to include in the degree of schedulability as DP:

$$\begin{aligned} DP(\mathrm{s}_i) \quad = \quad & \Big\{\tau^{f_b}_{ab/r_b} \in \mathrm{s}_i | \\ & \quad \Big(\neg\exists\tau^{f_c}_{ac/r_c} \in \mathrm{s}_i | \\ & \qquad (ab = ac) \wedge ((f_c = f_b \uplus ab) \vee \\ & \qquad \Big((f_b = f_c) \wedge \Big(R^w_{ab(f_b)(r_b)} < R^w_{ac(f_c)(r_c)}\Big)\Big) \vee \\ & \qquad \Big((f_b = f_c) \wedge (r_c < r_b) \wedge (R^w_{ab(f_b)(r_b)} = R^w_{ac(f_c)(r_c)})\Big)\Big\} \end{aligned} \tag{5.3}$$

and the final degree of schedulability becomes:

$$DOS(\mathcal{G}_i) = \begin{cases} c_1 = \dfrac{\sum\limits_{\mathrm{s}_j \in \mathcal{S}_i} w_j \sum\limits_{\tau^f_{ab/r} \in DP(\mathrm{s}_j)} \max\Big(0, R^w_{ab(f)(r)(j)} - D_{ab}\Big)}{|\mathcal{S}_j|} & \text{, if } c_1 > 0 \\ c_2 = \dfrac{\sum\limits_{\mathrm{s}_j \in \mathcal{S}_i} w_j \sum\limits_{\tau^f_{ab/r} \in DP(\mathrm{s}_j)} \Big(R^w_{ab(f)(r)(j)} - D_{ab}\Big)}{|\mathcal{S}_j|} & \text{, if } c_1 = 0 \end{cases} \tag{5.4}$$

where w_j is given by equation (5.2). Please notice that $R^w_{ab(f)(r)(j)}$ is the response time of process $\tau^f_{ab/r}$ in scenario s_j. Some processes that belong to several different scenarios might be counted several times. This is not a problem, but emphasises that a process belonging to several scenarios has a higher influence.

We will analyse the cost function for each of the two situations, c_1 and c_2. In the first situation, one or more processes do not reach their deadlines in at least one scenario. If this is the case, then we ignore all processes that meet their deadlines. The only objective is therefore to minimise the total lateness, meaning the total time that these processes violate their deadlines. While minimising these violations, some of the processes that satisfy their deadlines might see an increase in their response time. Once the configuration is schedulable, i.e. $c_1 = 0$, we will try to minimise the response times for all processes. In other words, the cost function will try to utilise the given resources in the best possible way in order to reduce the response times as much as possible.

In the following we will give two examples of calculating the cost function for two different policy assignments of the same application. The corresponding fault-tolerant process graphs are shown in Figure E.1 (Example 1) and Figure E.4 (Example 2) in Appendix E.1.2.

The calculations are shown in Table E.3 and Table E.4 in Appendix E.3. For both examples, the priorities, deadlines and the worst case execution times used by the response time analysis are included in the tables. Notice that elements defined by DP and therefore included in the summation, are given in bold.

From the results, we can see that since the degree of schedulability of both solutions is negative; it implies that all processes in all scenarios are schedulable. This can also be confirmed by looking at the response times found for each scenario and comparing these with the deadlines given. Since Example 2 has a lower cost value (-49) than example 1 (-43.3), then it is a better solution according to our definition.

Notice that our cost function is not the only possible approach. First of all, one could argue that when all deadlines are satisfied, we no longer need to minimise the response times. But having found a schedulable solution, it seems relevant to try to optimise the solution even further. One approach could be to reduce the power consumption. In that case, reexecution will be preferred over replication. Other approaches include minimising the communication or reducing the utilisation of the processing elements. Which approach should be chosen, depends on the requirements and therefore there is no general solution as such.

5.3.3 Choosing an Initial Solution

Before starting the optimisation, we must define an initial solution. The initial solution should be as close to the optimal solution as possible. The better an initial guess we make, the fewer iterations are needed for the optimisation to converge and the result is obtained faster. There are two approaches for creating such an initial solution. First approach is a general initial guess that is independent of the problem being solved. The second approach creates an initial guess only by looking at some of the parameters in the problem being analysed, but without actually evaluating the objective function. For our optimisation problem, it could for example be considering the ratio between processes and processing elements. If there is a relative high number of processing elements, then replication might be a good initial guess. A more advanced solution, as proposed in [23] could be to balance the utilisation between the processing elements.

Choosing an solution guess that depends on the problem and works well in most situations is not trivial and requires a lot of knowledge about the average system being analysed. Therefore, we choose the simpler initial solution approach. Thus, we assume that reexecution in most optimal solutions will be the most dominant policy assignment. Replication is rather expensive, as replicating a process will incur a fixed cost no matter whether the process fails or not. As a result, we will be able to tolerate more faults than necessary. Only the situations with enough hardware and relative short deadlines, replication will be most effective in order to satisfy the timing constraints.

On this background, we will choose reexecution as the initial policy assignment for all processes. Priorities will be the same as the original processes as defined in Section 5.1.

5.3.4 Choosing a Heuristic

Having defined the optimisation problem, the objective function and an initial solution, we need to choose which heuristic to use. There exists a number of different algorithms, among others:

- Hill Climber;
- Simulated Annealing;
- Tabu Search;
- Genetic Algorithms.

Each of these comes in different variants. Previously, different algorithms have been used in similar optimisation problems. In [14] and [23] Tabu Search was used, whereas [4] used Simulated Annealing. In some cases, even specialised meta-heuristics have been developed, such as the HOPA algorithm proposed in [18] used to find the optimal priority assignment. In that article, Garcia et al. showed that such a specialised meta-heuristic can outperform a more generic algorithm.

In order to keep our solution as simple as possible, we will use the Hill Climber algorithm. It has the advantage that it is very easy to understand and implement. In most situations the algorithm will also perform well.

The idea is that from the current solution the cost function is evaluated for all solutions in the neighbourhood. The best solution in the neighbourhood, as evaluated by the objective function, is chosen to be the new solution. The iteration continues from the new solution until there is no better solution in the neighbourhood. This approach is also called *steepest descent*, since we always take the solution from the neighbourhood that results in the largest change in the objective function. Another approach is to let the new solution be the first element in the neighbourhood that is better than the current solution. This implies that we do not have to evaluate the entire neighbourhood. If we know beforehand that only a few elements in the neighbourhood will be better than the current, it would be a better approach even though it might result in more iterations.

A drawback of the Hill Climber is that it might get caught at a local optimum as illustrated in Figure 5.7. Both Tabu Search and Simulated Annealing contain methods for avoiding being trapped at local optima. We accept this deficit as part of our solution, as we prefer a simple algorithm rather than a maybe better but more complex algorithm.

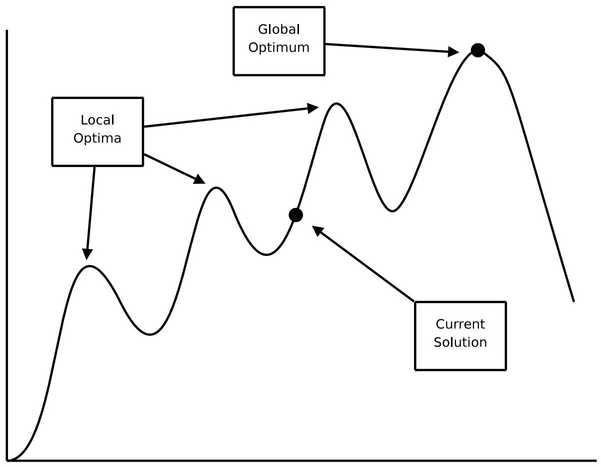


Figure 5.7: Illustrating that a Hill Climber can get stuck in a local optima [3].

The pseudocode for our implementation based on [36] can be seen in Algorithm 3.

Algorithm 3 Steepest Descent (Hill Climber Heuristic)

```
Require: bench should be a valid process graph
  for all $\tau_{ab} \in bench$ do
    apply reexecution to $\tau_{ab}$ {Add reexecution to all processes as the initial guess}
  end for
  repeat
    S0cost = DOS(bench)
    Scost = S0cost
    for all currentMove $\in$ getNeighbourhood(bench) do
      performMove(currentMove)
      currentCost = calculateDOS(bench)
      if currentCost < Scost then
        bestMove = currentMove
        Scost = currentCost
      end if
      undoMove(currentMove)
    end for
    if Scost < S0cost then
      performMove(bestMove)
    end if
  until Scost = S0cost
```

Chapter 6

Implementation and Testing

We have through the preceding chapters presented the theoretical background for doing design optimisation of fault-tolerant applications. In this chapter we will outline our implementation of the presented theory and how the implementation has been tested. The implementation is used to evaluate the behaviour of the proposed algorithms.

Java has been chosen as the programming language for our implementation. The primary reason for this choice is that Java provides a wide range of features to do a rapid development of software, such as support for object-oriented data model, automated memory management, a large library of functions to work with different data structures and portability across many platforms.

Section 6.1 gives details on the implementation of our program. It contains a description of our data model, which is used to represents a distributed embedded system with fault-tolerant applications. The description is supplied with UML class diagrams showing the relations between the data structures as well as the most important methods.

The implementation of the proposed response time analysis algorithm is explained in Section 6.2. In this section, the theory is linked with the actual methods. The chapter also contains some information about several optimisation strategies that were used to increase the speed of the implementation.

In Section 6.3 we describe the heuristics briefly with a primary focus on the extensibility of the implementation. We will present the approach used for testing our program in Section 6.4.

Finally, we provide a simple manual to the program in Appendix F.2 showing how the program can be used by a designer of embedded systems.

6.1 Design Overview

The implementation of the data structures is very similar to the concepts presented in the preliminaries. All entities in the model are represented as classes, for instance *Message* or *CommunicationChannel*. We have drawn several UML class diagrams for the main classes, which include the most important methods and attributes. The diagrams can be found in Appendix F.1.

We generalise processes and messages as *Activity*. This class groups their shared properties and methods. Classes *Process* and *Message* are derived from *Activity*, but each of them have some specific attributes. All activities contain information about their successors and predecessors. This is how we can model process graphs, which are represented by *Transaction* class. The attributes of *Transaction* include a list of activities that belong to that transaction, the period of transaction and other relevant data. Transactions are grouped into an *Application*, which is a part of *Bench*. A bench is top-level entity in our model. Besides the application, it encapsulates the hardware model given by the *Platform* class and the fault model given by the *FaultModel* class.

In addition to the data structures, we have a number of utility classes used to implement our algorithms as well as other auxiliary functionality such as a small GUI and a configuration parser. Notice that the GUI was created for displaying the FTPGs created by the program and has been extensively used throughout the implementation of the algorithms. All classes are grouped in the packages shown in Table 6.1.

SUN Java contains several very efficient implementations of sets and lists, which we extensively use to maintain collections of activities, fault-lists, fault-scenarios and so on. The utilised implementations include *HashSet*, *HashMap* and *Vector* to represent the collections of entities in the application.

The description of a system used as input to our program is done in XML. An example can be found in Appendix D.1, while the XML schema in Appendix F.3 formally defines the input file.

Package	Description
structures	Model classes
scheduling	WCDOPS+ algorithm and tools
heuristics	Hill-Climber algorithm and moves
launcher	Program launcher and GUI
config	Configuration reader (XML parser)
test	Experiments and unit tests

Table 6.1: The main Java packages in our implementation.

6.2 Implementation of WCDOPS++

The implementation of the response time analysis algorithm is inspired by an existing program called *aidalyze* [33], which is an RTA tool created by Redell. It supports several methods of doing response time analysis, where WCDOPS+ is one of them. However, due to our requirements and extensions we have decided to write our own code in Java based on fragments taken from [33]. It has also helped us to understand some of the inner workings of WCDOPS+.

Besides adding the necessary code to enable support for analysis of applications with multiple predecessors and conditional process graphs, we have also studied possible performance issues. We have found that the speed of the algorithm could be greatly improved compared to the implementation by Redell by introducing buffering of some of the data structures.

The purpose of the buffering is to cache data that requires much computational time to obtain, but once found, it will stay unchanged until the end of the analysis. An example of such data is an H-segment. A segment is a cluster of processes that have been grouped with respect to their priorities and mapping compared to the process under analysis. Since the segment is built using static information, we can store it in the memory and reuse next time we need it. This implies that the first iteration of the RTA will take about the same time as without buffering, but all subsequent iterations will be relative faster. As the number of iterations increase, the relative speedup will be increased even more.

The procedure of building segments (and sections) is based on recursive traversal of process graphs, which means that its execution time will depend on the number of process in the graph. For large graphs this operation will be very time demanding, and by using data buffering we reduce the required time to the minimum. This time can be reduced further, as follows. A segment is found by combination of two processes, τ_{ab} and τ_{ij}. Process τ_{ij} can be in the segment along with some other processes. But once we have such a segment, it can be cached for with all its processes, and not only τ_{ij}.

Here follows an example. Assume for τ_{ij} we find the H-segment $H_{ab}^{seg}(\tau_{ij}) = \{\tau_{ik}, \tau_{ij}, \tau_{il}\}$ and save it in the cache. The segment now can also be stored for processes τ_{ik} and τ_{il}. When the algorithm needs segment $H_{ab}^{seg}(\tau_{ik})$, the data buffer returns the previously found segment $H_{ab}^{seg}(\tau_{ij})$, because they are the same. This approach works also for H-sections, since they are built in the same way as segments. We have data buffering for other data structures as well, including priority and precedence relations, fault-scenarios and MP sets. We analyze the performance of the buffered RTA compared with non-buffered in Chapter 7.

The implementation of the algorithm consists of several parts, which are the algorithm itself (*WCDOPSPlusAnalyzer*), a class used to represent segments (*HSegSec*), and an additional class providing utility methods and the data buffer (*WCDOPSPlusToolsCached*). There are also other a couple of classes used to encapsulate intermediate results, such as branch interference (*BranchInterferenceResult*). They are only used as data containers and do not have any interesting functionality. The relations between the classes are shown in Diagram F.4 in Appendix F.1.

We have implemented the equations presented in Chapter 3 as methods in class *WCDOPSPlusToolsCached*. The relations between the main equations and the methods are given in Table 6.2.

Equation	Method
(3.5)	**int** phase(Activity ij, Activity ik)
(3.7)	**int** p0ijk(Activity ij, Activity ik)
(3.24)-(3.28)	**int** updateJitterAndOffset(Activity ab, **int** updateFlag)
(3.43)-(3.47), Pseudocode 4	BranchInterferenceResult BranchInterference_is(Activity ab, Activity ac, Activity ik, Activity ir, **int** p, **int** w)
(3.13)	**int**[] Wik(Activity ik, Activity ab, Activity ac, **int** w)
(3.15)	**int**[] WstarI(Transaction i, Activity ab, Activity ac, **int** w)
(3.15)-(3.17)	**int**[] Wstar(Activity ab, Activity ac, **int** w)
(3.18)	**int** Wabc(Activity ab, Activity ac, **int** pab, **int** w)
(3.20)	**int** Labc(Activity ab, Activity ac)
(3.21)	**int** pLabc(Activity ab, Activity ac)
(3.22),(3.49)	**int** Rabc(Activity ab, Activity ac)
(3.22),(3.49)	**int** Rwab(Activity ab)
(3.48)	**boolean** canCoExists(Activity ab, Activity ij)

Table 6.2: Relations between the main equations of the response time analysis as defined in Chapter 3 and the implementation.

6.3 Implementation of the Heuristics

In this section, we will briefly explain the implementation of the heuristics. In Figure F.3 in Appendix F.1 the class structure of the heuristics is shown.

The implementation is held as extensible and modular as possible. This way, it will be easy to replace either the chosen heuristic, the definition of the neighbourhood, the objective function or add additional moves. As a result, a number of interfaces have been defined in Table 6.3.

The *MoveInterface* is implemented by the two moves described in Section 5.3.1, *FaultToleranceMove* and *RemapMove*. The interface defines two primary methods, *performMove* which executes the move while *undoMove* reverses to the state before the move. This enables the heuristic to evaluate each move in

Interface	Methods
Move	**void** performMove() **void** undoMove()
CostFunctionInterface	CostValue calculateCostValue(Bench bench)
HeuristicsInterface	HeuristicResult optimize(Bench bench, **int** maxIterations)

Table 6.3: Interfaces in the part of the implementation covering heuristics. Notice that the classes *CostValue* and *HeuristicResult* are just datacontainers.

the neighbourhood seperately. For each move defined by the neighbourhood, *perfomMove* is called, the costfunction is evaluated with the new configuration and compared with the previous results, and finally the *undoMove* method is called to reverse the move.

The two other interface, *CostFunctionInterface* and *HeuristicsInterface* should be self explanatory.

The neighbourhood locates, for the current configuration, all possible moves as defined above. It is implemented as an *iterator* in Java, making it rather flexible while the generation and iteration itself can be expressed very compact:

```
for (Move move : new Neighbourhood(bench)) {
   ...
}
```

6.4 Tests

In this section we will present the approach used for testing our program. In this context, testing is about verifying that the implementation is correct and for all possible inputs it produces the correct results as defined by the theory. Notice that testing is not about performance evaluation such as speed or memory consumption, but only the correctness of the program. Nor does it test, whether the theory is correct, only that the program implements the theory correctly.

The tests, which all have been scripted as automated tests, are based on two different strategies. The first strategy it to use constructed tests that verify very specific situations. These tests has been used extensively when testing our extensions to the existing response time analysis. By first calculating the expected result by hand, we are able to verify the result obtained by our implementation. The same approach has also been used when testing the graph as well as the heuristics.

The second approach is sanity testing for random applications. For a large number of randomly generated applications, we verify with a number of sanity requirements that the implementation produces correct results. This strategy has been used for the response time analysis as well as the graph algorithms. The sanity checks for the FTPGs are shown in Appendix G.1. The generation of the random application will be explained in subsection 6.4.3. Furthermore we will in subsection 6.4.2 explain how Redells existing implementation have been used with the same purpose.

We have covered only a limited number of tests cases, and therefore the implementation cannot be considered fully tested. More comprehensive testing is always desireable, but was not feasible due to the timing limitations. We have however tested all critical functions used across different parts of the program.

The implementation of the tests is located in the packages *test*.∗ as part of the enclosed source.

6.4.1 Component Tests

The goal of component testing (also called *unit tests*) is to evaluate each component or *unit* of the program independently from other components, which allows to start testing at very earlier stages of the development. The tests have been performed using JUnit [28].

Following components of the program have been tested using unit tests:

- **Primary data structures** such as *Application*, *Activity*, *Transaction* and others.
- **Operations on FTPG** are both tested on simple test cases and by doing some stress tests. This approach is described below.
- **RTA Algorithm** black-box tested to produce correct results with a number of composed manually applications. These tests were constructed such

a way that they should expose errors related to our extensions.

- **Utility methods used in RTA** such as producing different grouping of processes (XP, MP, HEP etc) and the buffering of data structures.

- **Operations on segments and sections** used for locating segments and sections both in regular and fault-tolerant applications, computing precedence and blocking relations between segments.

- **Optimisation Heuristic** including the cost function and the neighbourhood generator.

6.4.2 RTA Comparisons

Yet another testing technique was used to ensure that our implementation of the response time analysis is correct. As the problems that we are solving are NP-complete, we are not able to find an exact solution to WCRT. Therefore it is practically impossible to test against the exact solution that our solution indeed is correct.

Another approach would be to compare our results against other algorithms or even approaches to response time analysis, but the results might not be trustworthy. If our response time analysis finds a lower worst case response time than another algorithm, then we would not know whether the result is less pessimistic or actually wrong. Only if our solution returns a higher worst case response time, we would know that our algorithm is correct[1]. But this would unfortunately also imply that our solution in that case is more pessimistic.

We have chosen to compare our version with the original implementation of WCDOPS+, *aidalyze*, made by Redell [33]. It is done by generating random graphs as explained in subsection 6.4.3 and performing the analysis with both implementations. The upper bounds for response time are than compared for each process.

We are not able to compare the results for graphs with multiple predecessors, because aidalyze is not capable of this feature. It means that we have only done automatic testing of our algorithm with applications having single predecessors.

[1]If assuming that the other algorithm is correct

6.4.3 Producing Synthetic Applications

The synthetic applications used in our tests were generated by the tool called Task Graphs for Free (TGFF v3.0) [12]. We have only used the new algorithm in TGFF called "Series-parallel". The basic configuration that we have used for both tests and the evaluations can be seen in Appendix G.2.

In Table 6.5 we have shown and explained the primary parameters which are used in the configuration file.

Variable	Description
seed	An arbitrary number to initialise the pseudo-random number generator. Should be different for each run in order to ensure that the generated graphs are different. We *java.util.Random* of Java to generate a random number
tg_cnt	Number of task graphs to create. Always set to 1
task_cnt	The number of processes in the generated application. Please also see the discussion below.
series_must_rejoin	True or false. If true, then graphs with multiple predecessors are created.

Table 6.5: Explaining the important variables in the configuration file for *Task Graph For Free.*

In order to obtain stable results with our evaluation experiments, we need to generate a number of random process graphs with the same number of processes for every case and take an average of that. Unfortunately, it cannot be done in a simple way with TGFF. The program accepts a minimum number of processes per process graph only, so the actual number of processes may be close to the minimum, but not exactly the same. Therefore a workaround has been used to bypass this limitation.

The idea, which is also given as pseudocode in Algorithm 15 in Appendix C, is to continue generating graphs until the required amount with the exact number of processes is collected. Assume that we want a graph with 10 processes ($rP = 10$) and 5 graphs ($nP = 5$). We start by requesting a graph, G, from TGFF with 10 processes, and we get a graph with 12 processes instead. Then we set $aP = 12$ and G is added to the set of final graphs (sG).

We continue requesting graphs with 10 processes, but only adding the ones that have the same number of processes as the first (12 processes), until we have the desired number of graphs. sG will therefore only contain graphs with the exact same number of processes.

Chapter 7

Evaluation

In this chapter we present and elaborate on the results obtained through experimental evaluation, including a realistic case study taken from the automotive industry. The results from this evaluation can be found in Section 7.2.

The idea of the evaluation is generally to analyse the performance of the different algorithms. In contrast with the tests, where only the correctness of the results was important, the evaluation aims at determining other properties of our algorithms. This includes the speed, memory consumption and the quality of the results obtained by the algorithms.

Two major components of our system are to be evaluated. The first one is the response time analysis algorithm. We will compare different approaches of response time analysis on fault-tolerant process graphs and evaluate, how fast and precise they are. The reason for this separated evaluation is that the response time analysis also can be used in other perspectives than fault-tolerance policy assignment. Therefore it is interesting to have an independent assessment of the RTA algorithm.

The second component is the heuristic algorithm used for the optimisation of the fault-tolerance policy assignment. It will be evaluated using different randomly-generated synthetic applications in order to determine the quality of the selected approach and its suitability for the policy assignment and mapping problem.

The experiments have been done on a cluster consisting of machines running Linux and equipped with “Dual Core AMD Opteron Processor 175” (2200 MHz) and 2 GB of memory. The job distribution among the nodes is controlled by an internal job dispatcher. Since the machines are dual-core, the job dispatcher allows two parallel jobs to be running simultaniously on each machine. This has two side-effects. First that the 2 GB of memory are shared between the two experiments, leaving only 1 GB for each job. Secondly, we notice that the response time algorithm is more memory intensive than CPU intensive. Since the memory and the memory access are shared between the two processes, the memory access might be a bottleneck. This might result in a performance penalty. This has, however, not been investigated further.

7.1 Synthetic Applications

In the first part of the evaluation, we will test our algorithms with a number of synthetic applications. The reason for using synthetic applications is the lack of real-life examples to run the tests on. The synthetic applications should represent realistic applications and enable us to study the behaviour when changing different parameters of the experiments.

In all cases when synthetic applications have been used, the result is an average over ten different experiments. In a few cases we have not been able to obtain results from all ten experiments, and have instead averaged the results available. In Figure 4.9 and 4.8 we have shown that the number of processes and scenarios, respectively, depends very much on the topology of the graph. As the evaluations depend very much on these numbers, we average to ensure that we get a representative spectrum of different topologies. Another issue is related to the Java VM. As running a program requires the virtual machine to warm up, i.e. allocating memory and compiling into native code, before the performance is stabilised, we reduce this effect by running a number of consecutive experiments. One should also notice that the timing for the different experiments is not precise, primarily due to the non-deterministic behaviour of the memory management in Java controlled by the garbage collector.

The syntetic applications have been created as discussed in Section 6.4.3. We have used four processing elements, and the best case execution time is selected randomly between 10 and 20, while worst case execution time is between 15 and 25. The BCET is always chosen such that it always is equal or less than the WCET. The period is constant and large (1000000) and the deadlines are equal to the period. To simulate that some mappings are not allowed, we set a probability of 10% for a given mapping not to be allowed. The priority is chosen randomly between 10 and 20.

Notice that all plots unless otherwise stated are logarithmic. The reason being that most of our results grow exponentially with the size of the problem, is more easy to interpret the results with a logaritmic plot.

7.1.1 Buffering Versus no Buffering

We start by evaluating the effects of introducing buffering as described in Section 6.2. Recall, that the basic idea was to avoid performing identical computations more than once. It includes, for example, locating H-segments and H-sections, which are static information that does not change across iterations. We would therefore like to evaluate how much the execution time is decreased when using this buffering compared to the naive solution. The result is shown in Figure 7.1.

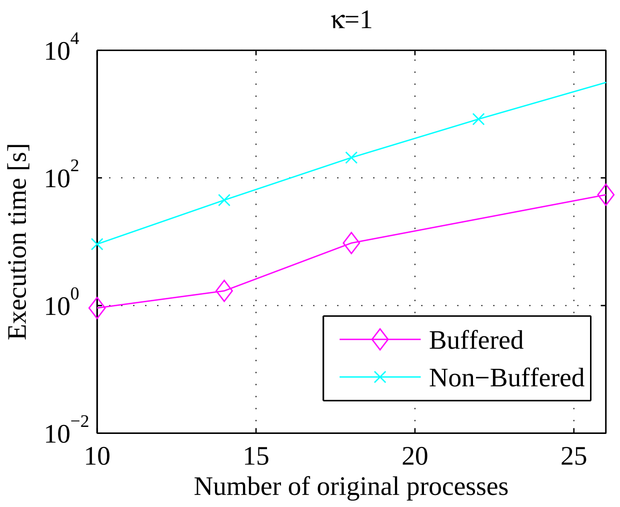


Figure 7.1: Execution times for the RTA with and without buffering for different number of processes and $\kappa = 1$.

The diagram above shows the behaviour of the algorithm for the growing number of processes (the x-axis), when κ is kept constant. The y-axis represents the execution time of the RTA. We can see that the execution time of the analysis depends on the amount of the processes. Both with and without buffering the execution time grow exponentially with the number of processes. It can also be seen that the buffering drasticly reduces the execution time of the algorithm with a factor between 10 and 100. It could also seem as the relative difference between the two approaches increases with a higher number of processes, although the tendency might not be certain.

Since the buffering approach is clearly faster than without the buffering, it has been used in all further experiments.

7.1.2 Evaluation of Response Time Analysis Approaches

In this section, we will evaluate the three different approaches of doing conditional response time analysis as introduced in Section 3.3: *IC*, *CS* and *BF*. We will analyse the different approaches for different number of processes and different number of faults, and try to capture any trends in the experiments.

First we will consider the execution times of the three approaches. In Figure 7.2 the execution times are illustrated for each approach for $\kappa = 0$ and $\kappa = 1$. As expected, the execution time are more or less identical for $\kappa = 0$, since the amount of calculations are the same. Any divergence is explained by differences in the random topologies of the graphs that are not completely balanced out by taking the average of 10 experiments. The execution time is exponentially increasing for all three approaches as the number of processes increases. Looking at $\kappa = 1$, we see that for all three approaches the execution time also increases exponentially with the number of processes. It can also be seen that IC takes considerable longer than CS and BF with a factor between 10 and 100. CS and BF is on the hand comparable, but with CS slightly slower than BF.

In Figure 7.3 we have plotted the execution time for a fixed number of processes (10) for the different approaches and an increasing number of faults. Here we can see that the execution time of all three methods increase exponentially with the number of faults, but IC increases faster than the others. It also seems that the relative difference between CS and BF increases as the number of faults increases.

In Figure 7.4 we show the execution time for CS and BF versus the number of processes in the fault tolerant process graph. Since these will have different numbers it is not possible to average these values. Notice that the x-axis is logarithmic. It can be seen that CS and BF are relative close to each other, but again with CS being slightly slower than BF. IC again increases much faster than CS and BF.

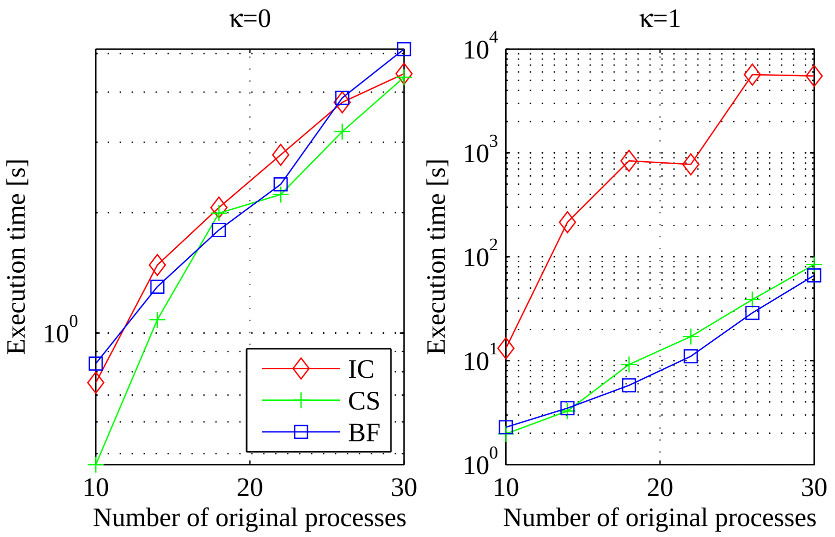


Figure 7.2: Execution time for the three different approaches for RTA: IC, CS, and BF. Notice that execution times for $\kappa = 0$ are more or less identical while IC is much slower than CS and BF for $\kappa = 1$

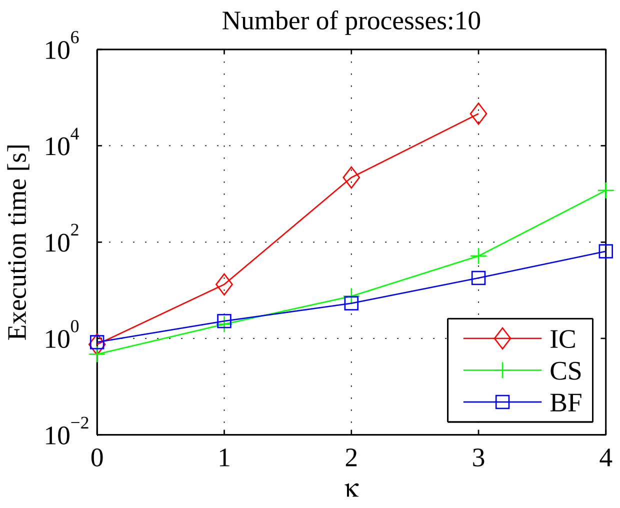


Figure 7.3: The execution time for the three different approaches for RTA with an increasing κ. Number of processes is 10.

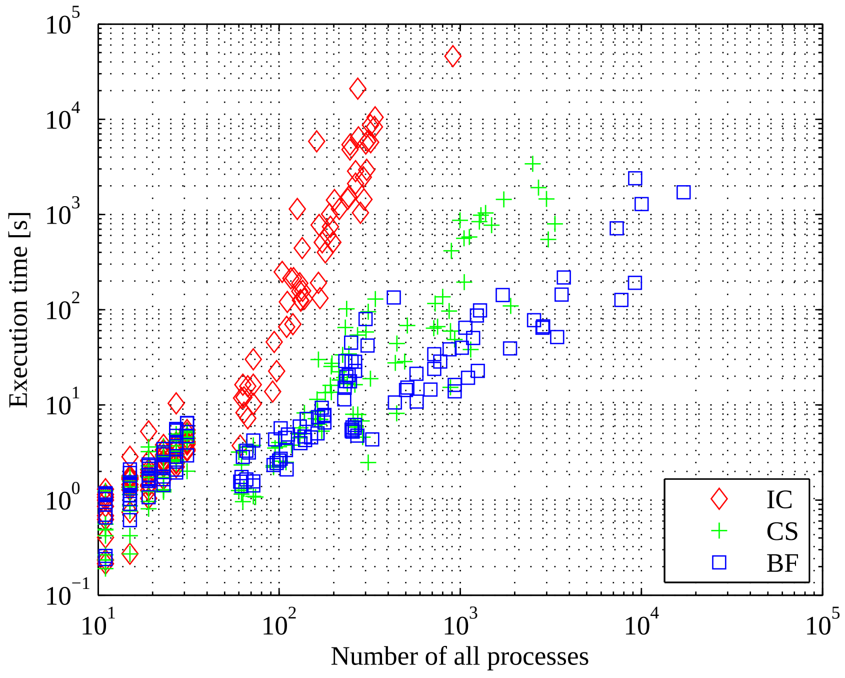


Figure 7.4: Execution time for the three different approaches for doing versus the number of processes in the fault-tolerant process graph.

The behaviour of IC in these figures was expected. Recalling that the response time is computed by considering all cases of possible process interferences, it is trivial to realise that for IC, the interference analysis includes all process in the conditional process graph regardless of their scenarios. Therefore it is not practical to use our RTA algorithm for doing analysis with IC, even in cases when just pessimistic bounds on response times are enough.

The other two methods look very similar, which contradicts our original hypothesis that the CS approach to be less precise than BF, but faster. However, the results demonstrate a contrary situation. By looking at the execution time plots, we can conclude that BF analysis is actually faster than CS. The cause for this behaviour is most likely found in the nature of the algorithm. When doing conditional analysis in the CS approach, a great amount of time will be spent on computing interference for processes that belong to all scenarios, for instance the root process.

Another example was illustrated in subsection 3.3.2, where a given process was affected by mutually exclusive processes. Even though the situation is not realistic, the algorithm must include all these processes in its consideration. When the number of processes or faults increases, the number of mutually exclusive processes increases too thereby prolonging the execution time. It also confirms our discussion about the size of fault-tolerant graphs in Section 4.7, since WCDOPS+ depends very much on the total number of the processes in applications rather than fault scenarios.

That might also explain why the relative difference to the execution time of BF increases with an increasing number of faults. Of course, the overall speed is still much higher than IC, since most processes are eliminated in the outer loop. In order to optimise the speed of CS, the algorithm must be analysed further such that the situations above, if possible, can be avoided.

In the second part our evaluations of the response time analysis approaches, we look at the degree of schedulability (DOS) in order to consider how precise the three approaches are. Please notice what is shown in the following plots, is not exactly DOS as defined by equation (5.4), since the deadline has been left out when doing the calculation. The reason is that we, as described in subsection 3.3.1, are not able to create realistic deadlines and assume that the deadline is equal to the period for all processes. Since the deadlines therefore are much larger than the response times, any differences in the response times for the different approaches would be more or less concealed by the deadlines. As a result, we have chosen to leave out the deadlines, such that the DOS shown resembles a weighted sum of the response times. This will not affect the comparison between the approaches, since the processes share the same deadline.

We consider BF as being the exact solution[1] because it considers each scenario seperately. Any DOS higher than the DOS found by BF, is a result of increased pessimism. In Figure 7.5 the plots show for $\kappa = 0$ and $\kappa = 1$, how the DOS varies for the number of processes in the application. Please notice that the number of processes comes from the initial application, not the fault-tolerant process graphs and that the figure is not in a logarithmic scale. For $\kappa = 0$ the DOS is more less identical for all three methods. This is, as with the execution time, expected since the calculations are identical for all three approaches. For $\kappa = 1$ the situation is somewhat different. It can be seen that while BF only increases slowly, IC increases almost exponentially. It means that the more processes, the more inaccurate IC is. The reason is that the FTPG simply contains more processes that all are able to interference with each other, and the response time will increase exponentially with the number of processes. CS increases also

[1]When keeping in mind that the RTA is not exact. A more precise result can only be obtained by improving the precision of the RTA.

faster than BF, but is relatively much closer to BF than IC. From this we can see that CS is more accurate than IC, and provides a relative acceptable estimate of the DOS. The reasons for the difference in the DOS between BF and CS was explained in subsection 3.4.

In Figure 7.6 the DOS is shown for a fixed number of processes (10) and an increasing number of faults. As expected, we see that the DOS for IC increases exponentially, whereas the DOS for CS and BF decreases slowly. At first this might seem surprising that they decrease, but it is due to the fact that with an increasing number of faults, the number of scenarios also increases. Recall that the DOS is calculated as an average since we divide it by the number of scenarios. For each scenario, the DOS is a sum over the processes times a weight, which is inversely proportional to the number faults. It implies that the additional scenarios added when introducing more fault, actually decrease the total DOS.

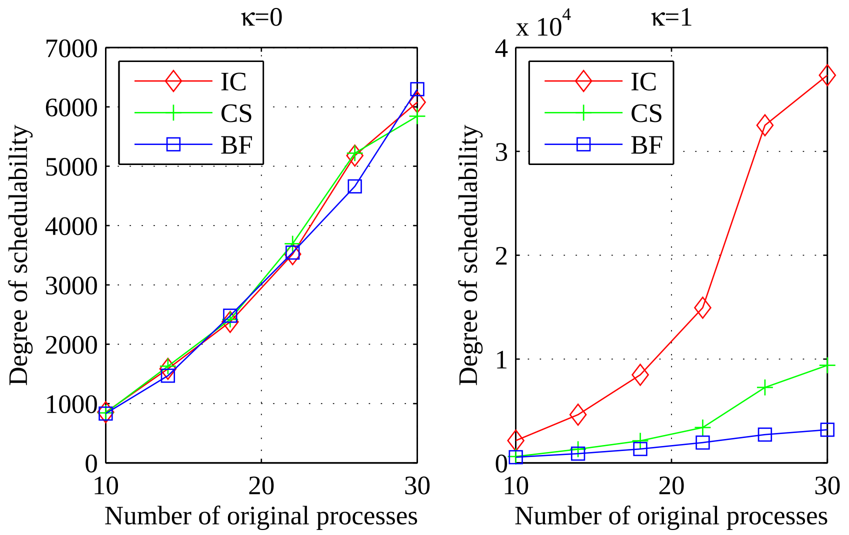


Figure 7.5: The degree of schedulability for the three different approaches for RTA.

Based on the presented results, we have decided to use the BF approach in the fault-tolerance policy optimisation heuristics. This approach finds the least pessimistic boundaries on the response time and has the best performance among all others.

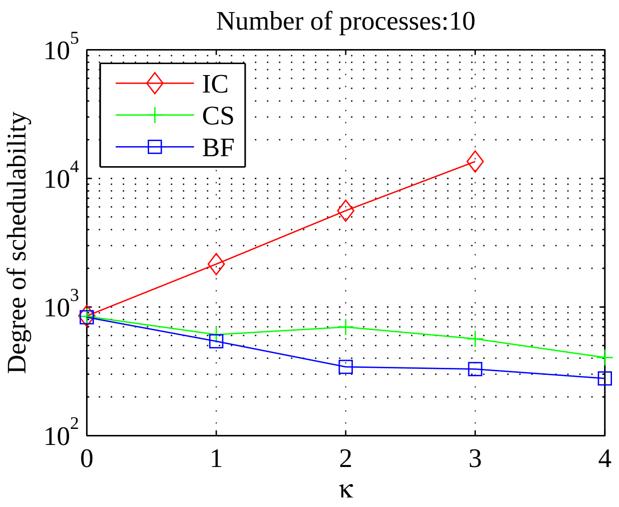


Figure 7.6: The degree of schedulability for the three different approaches for RTA with an increasing κ. Number of processes is 10. Notice that IC has not been calculated for $\kappa > 2$, but we strongly assume that the trend will continue.

7.1.3 Convergence and Evaluation of Heuristic

In order to test our algorithm, we consider the example given in Figure 7.7 and Table 7.1 where $\kappa = 1$. We apply reexecution and replication to all processes as the initial guess to test that it converges to the same solution.

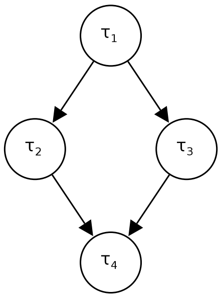

$$\begin{tikzcd}
 & \tau_1 \arrow[dl] \arrow[dr] & \\
\tau_2 \arrow[dr] & & \tau_3 \arrow[dl] \\
 & \tau_4 &
\end{tikzcd}$$

Figure 7.7: A Diamond Shaped Application

In Appendix E.4 the convergence is shown for each of the two initial guesses. It can be seen that the two situations, as expected, end up with the same

τ_i	P_i	D_i	$\mathcal{M}(\tau_i)$	N_0	N_1	N_2
1	4	9	N_0	5	6	6
2	2	20	N_2			2
3	3	15	N_0	3	12	2
4	1	20	N_0	4	15	3

Table 7.1: Properties of the Processes in Figure 7.7. Best case execution times are equal to worst case times.

cost value. The final configurations are, however, not exactly the same. The policy assignments are identical, but the mappings are not same. It illustrates that there might be more than one solution with the same value of objective function, such that they will be considered equally good. The found solution depends much on the direction in the solution space which the heuristic arrives from. This is partly controlled by the initial guess, but also by the choice of heuristic.

Please also notice that for both initial guesses the initial solution is not schedulable. This is indicated with the positive initial cost. In both situations the resulting configuration is schedulable which can be seen from the final cost, -22.

7.2 A Real-Life Example with a Cruise Controller

In order to evaluate our results on a real-world example, we test our technique on a model of cruise controller taken from [38]. We assume that the system should be protected against two faults, $\kappa = 2$. However, due to missing information about several system constraints, a number of assumptions have been made to this model.

In Figure 7.8 the process graph has been shown, and the properties of the processes can be found in Table 7.2. Furthermore, the initial mappings as given by [38] are shown in Figure 7.9. Please notice that since our response time analysis requires that the process graph has a single root, we have added a dummy proces, τ_0.

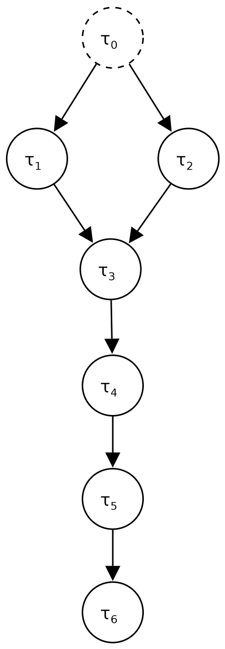


Figure 7.8: Process graph of the Adoptive Cruise Controller.

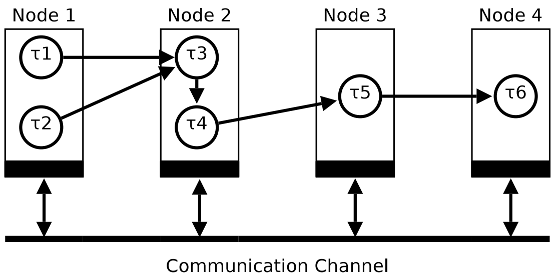


Figure 7.9: Initial mapping of the processes for the Adaptive Cruise Controller.

The problem is now, that we are missing information about the priorities of the processes as well as the worst- and best case execution times on other nodes than the ones initially assigned. The WCET and BCET given in Table 7.2 are only for the initial mappings. In order to make a qualified guess of the different mappings, we consider the purpose of the different processes, which is described in Table 7.3. Processes τ_1 and τ_2 reads values from some sensors while τ_6 controls an actuator. We therefore assume node 1 is the only node physically connected to the sensors, while node 6 is the only connected to the actuator. As a result, processes τ_1, τ_2 and τ_6 cannot be remapped which is reflected in the mapping table (Table 7.4). Since the purpose of remaining processes are just calculations, we assume that these can be remapped to all other nodes. Since the computational processes, τ_3, τ_4 and τ_5 are initially assigned to node 2 and 3, we assume that these are twice as fast as node 1 and 4. We model this by multiplying the BCET and WCET of the processes accordingly in the mapping table. We let all priorities be 1. The choice of priorities is not important, since there is almost no parallelism in the system. It will therefore, except for τ_1 and τ_2, be the precedence constraints that determine which in which order the processes are run.

Process	BCET [ms]	WCET [ms]	Deadline [ms]	Period [ms]	Node
τ_0	0	0	20	20	1
τ_1	2	2	20	20	1
τ_2	2	2	20	20	1
τ_3	2	6	20	20	2
τ_4	2	2	20	20	2
τ_5	2	6	20	20	3
τ_6	2	2	20	20	4

Table 7.2: Properties of the processes in the Adoptive Cruise Controller.

Process	Purpose
τ_0	Dummy root process
τ_1	Measures the velocity of the vehicle
τ_2	Measures the distance to the closest vehicle in front
τ_3	Calculates the relative speed to the vehicle in front
τ_4	Calculates the desired velocity
τ_5	Given the desired velocity and with information about the engine, it calculates the absolute value of the throttle
τ_6	Controls the physical changing of the throttle

Table 7.3: Purpose of the different processes in the Adaptive Cruise Controller.

Since we add fault tolerance, we are also able to accept that the period and deadline, both given as 20 in [38], is increased to 30 for all processes.

Process	N_1	N_2	N_3	N_4
τ_0	(0, 0)	(0, 0)	(0, 0)	(0, 0)
τ_1	(2, 2)			
τ_2	(2, 2)			
τ_3	(4, 12)	(2, 6)	(4, 12)	(4, 12)
τ_4	(4, 4)	(2, 2)	(4, 4)	(4, 4)
τ_5	(4, 12)	(4, 12)	(2, 6)	(4, 12)
τ_6				(2, 2)

Table 7.4: Mapping table of the Adaptive Cruise Controller where execution time is given as BCET/WCET in [ms].

The XML-file describing the system is given in Appendix D.1. Running our program gives the result shown in Appendix D.2. From the results, we can observe a number of things. The initial degree of schedulability after having added reexecution to all processes, is negative (-2328) and therefore the system is already schedulable in its initial configuration with reexecution. After 4 iterations the degree of schedulability it improved to -2800. The result says that τ_1, τ_2 and τ_6 should be reexecuted while the other processes should be replicated.

Notice that the model is not completely realistic for at least two reasons. First of all, all processes except τ_6 should pass on their result on to the succeeding process. This implies that some messages must be included in the model to simulate the time when transfering the messages on the bus. Even though messages are included, it is not very clear in [38] how these are modelled and there are not given any transmission times for these messages. As a result, we have chosen not to include the messages. The second deficit of the example, is that the cruise controller will most likely be implemented on a system together with other components. As a result there will be other processes, possible with a higher priority, mapped to the same nodes, as well as other messages on the bus. This would create preemptions on the processing elements, and the response times of the processes and messages might therefore be prolonged. We have, however, not tried to include this in the system, since we would risk creating completely unrealistic situations. Secondly, this would for the current solution imply creating the hypertransaction with the accompanying deficies as described in subsection 3.3.1.

Chapter 8

Conclusions and Future Work

In this thesis we have presented an approach to design optimisation of safety-critical fault-tolerant embedded systems. We have considered hard real-time systems with event-triggered scheduling and a distributed architecture. The applications are modelled as processes with control and data dependencies grouped into process graphs. In order to protect processes against a fixed number of transient faults, we have used two fault-tolerance techniques – active replication and reexecution.

Fault-tolerant process graphs. In order to represent fault occurences in an application we have used fault-tolerant process graphs. It has been shown that existing approaches to work with process graphs are not completely suitable to our problem. Therefore we have proposed a new notation and algorithms that can be used to build and modify such graphs for any number of faults. In addition we have derived equations to compute the bounds for the number of processes and scenarios in the graphs.

We have only considered applications where all processes are protected against the same number of faults, but the developed algorithms can also be extended to include different numbers of faults for each process.

Response time analysis. We have based our response time analysis on the existing WCDOPS+ algorithm. This algorithm has been chosen as being state-of-the-art in the area of response time analysis for event-driven distributed systems with data dependencies. It also includes support for messages modelled as non-preemtive processes.

One contribution of this thesis is the extension of the algorithm in order to support applications where processes can have several predecessors. It has been shown what parts of the algorithm must be updated to reduce the pessimism and obtain more correct results. The major contribution has been an extension to support conditional analysis on an fault-tolerant process graph. Only applications with a single transaction have been considered.

The algorithm has been used to evaluate the system while doing fault-tolerance policy assignment. During the evaluations we have concluded that the performance of the algorithm is better when applications are analysed using a brute force approach. Therefore the brute force approach has been selected as the response time analysis for the policy assignment and mapping.

Mapping and fault-tolerance policy assignment. The purpose of fault-tolerance policy assignment is to determine which fault-tolerance technique should be assigned to the processes in a safety-critical application. The selected assignment should guarantee that for any combination of faults the application can obey its timing constraints.

We have applied the Hill Climber algorithm to derive the policy assignment and mapping. The heuristic is driven by the results provided by the response time analysis. In order to utilise these results in the most optimal way we have defined an objective function based on the timing constraints of the application. The objective function also depends on the likelihood of each fault scenario. We have assumed that scenarios having many faults are less possible than scenarios with fewer faults, but this weighing can be improved if the designer knows more details about the system.

The objective function is defined in such a way that it will direct the heuristic to a schedulable solution as the primary goal. When it finds a policy assignment that is schedulable, the objective function will help to reduce the response times and thereby choose the least costly configuration.

Evaluations. The results of the evaluation have demonstrated the performance of our methods. It has been shown that the pessimism of the response time analysis is significantly reduced compared to the approach where the fault conditions are ignored. However, the execution time of the algorithm with condition separation has been reported to be larger than for the brute force version. It means that the approach producing the most exact results is also the fastest one. The conclusion is that there are still parts of the algorithm that can be optimised in order to reduce the execution time of the version with condition separation.

The evaluation of the proposed approaches has been done with numerous synthetics applications. We have also used our method to optimise policy assignment for a real-life example from automotive industry, a model of a cruise controller. The experiment showed that the optimisation heuristic is able to create and improve the schedulability while tolerating a fixed number of faults.

The design optimisation techniques presented in this thesis are all based on several assumptions about the system architecture and application model. These assumptions were necessary to reduce the complexity of the problem, but on the other hand they brought the model to an abstraction level that is not directly applicable in a practical context. Therefore this thesis can be used as a foundation for further research on the topic, and we offer a number of important improvements in the following section.

8.1 Future Work

The last part of this thesis considers several suggestions for future research. In order to put our work into perspective we will also discuss how to make our methods more valuable from the point of view of a designer.

Applications with several transactions. The presented approach will only be useful for applications with a single transaction, which is very unlikely in the real world. Therefore it would be obvious to extend the presented methods to support applications with several transactions. Such modifications will require that both the fault model and the response time analysis are extended.

Performance optimisation. It has been shown that the response time analysis with condition separation produces results that are very close to the brute force solution. However, the last approach requires additional work to be done such as splitting graphs into scenarios. An improvement of the RTA algorithm would be to find and eliminate the bottlenecks thereby reducing the time needed to find the response times using condition separation. This might bring the execution time below the BF approach, while still obtaining rather precise results.

Policy assignment optimisation. The policy assignment is done by a Hill Climber based heuristic, which is very simple and has several weaknesses. As a part of future work, one can try to combine other, possible specialised, heuristics that are more tuneable and can escape from local minima. Another approach would be to introduce delta evaluations into the heuristics. With delta evaluations one only calculates the parts that are different between neighbouring solutions. The reason being that the the response time analysis for two solutions in a given neighbourhood, will be very close to each other. Therefore it should be possible to reuse at least some of the results across the different solutions.

Other optimisations might also involve that the fault model is extended with additional information such as different probability of having fault on different processing elements. This information could be used to adjust the weights used in the cost function and in this manner model more realistic behaviour.

Usability of the method. With the presented approach it is possible to find an optimal policy assignment if there exists one. However, the method will not be able to help the designer, if the system is not schedulable or the desired level of fault tolerance can not be obtained with the given system configuration. For such situations an improvement could be to reuse the available information about the system in order to figure out what parts of the hardware and possible of the application that should be changed in order to meet the requirements.

Another interesting option for future work would be to integrate the proposed implementation into an existing tool for design analysis and optimisation.

Appendix A

List of Notations

Application Architecture

$\mathcal{A}$	Application. The set of transactions
Γ_i	Transaction. A set of processes that are related through control or data dependencies
T_i	Period of transaction Γ_i
$pred(\tau_{ij})$	Set of predecessors to process τ_{ij}. Can be either processes and/or messages
$succ(\tau_{ij})$	Set of successors to process τ_{ij}. Can be either processes and/or messages
D_{ij}	Deadline of process τ_{ij}

Process

τ_{ij}	Process τ_j in transaction Γ_i
$\mathcal{M}(\tau_{ij})$	The processing element on which process τ_{ij} is mapped
C_{ij}^b	Best case execution time of process τ_{ij}
C_{ij}	Worst case execution time of process τ_{ij}
P_{ij}	Priority of process τ_{ij}

Message

m_{ij}	Message m_i in transaction Γ_i
$\mathcal{M}(m_{ij})$	The communication channel on which message m_{ij} is mapped
C_i^{mo}	Worst case transmission time (WCTT) of message m_i
C_i^{mo}	Best case transmission time (BCTT) of message m_i
C_i^{m}	Worst case transmission time (WCTT) of message m_i when considering mapping of sending and receiving processes in current configuration
C_i^{m}	Best case transmission time (BCTT) of message m_i when considering mapping of sending and receiving processes in current configuration
P_i^{m}	Priority of message m_i

Hardware Architecture

N_i	A processing element (PE) in the hardware architecture

Scheduling

Φ_{ij}	Dynamic offset of process τ_{ij}
φ_{ij}	Phase of process τ_{ij}
J_{ij}	Dynamic jitter of process τ_{ij}
$J_{ij}^{seg}(\tau_{ab})$	Dynamic jitter of H-segment $H_{ij}^{seg}(\tau_{ab})$
$hp_i(\tau_{ab})$	Set of processes from transaction Γ_i with a priority higher or equal to process τ_{ab} and on the same PE as process τ_{ab}
$lp_i(\tau_{ab})$	Set of processes from transaction Γ_i with a priority lower than process τ_{ab} and on the same PE as process τ_{ab}
B_{ij}	Blocking time of process τ_{ij} due to non-preemptible lower priority processes (messages)
R_{ij}^{b}	Best case response time of process τ_{ij}
R_{ij}^{w}	Worst case response time of process τ_{ij}
t_c	Critical instant
n_{ijk}	Number of pending instances of process τ_{ij} at critical instant t_c
$H_{ij}^{seg}(\tau_{ab})$	H-Segment
$H_{ij}(\tau_{ab})$	H-Section
w_{ij}	Length of the busy period for process τ_{ij}

Fault-Tolerant Process Graph

$\mathcal{G}_i$	Fault Tolerant Process Graph
s_{is}	A scenario: a trace through graph $\mathcal{G}_i$ for a certain combination of conditions
$\mathcal{S}_i$	The set of scenarios for graph $\mathcal{G}_i$
$\tau^f_{ij/r}$	An element of the FTPG; process τ_{ij} with faultlist f and replica number r
τ^f_{ij}	Equivalent to process $\tau^f_{ij/0}$
$m^f_{ij/r}$	An element of the FTPG; message m_{ij} with faultlist f and replica number r
m^f_{ij}	Equivalent to message $m^f_{ij/0}$
$succOrg(\tau^f_{ij/r})$	The set of processes corresponding to $succ(\tau_{ij})$
$predOrg(\tau^f_{ij/r})$	The set of processes corresponding to $pred(\tau_{ij})$
$\mathcal{P}_x$	The set of processes with reexecution as fault tolerance
$\mathcal{P}_r$	The set of processes with replication as fault tolerance
$RS(\tau_{ab})$	The set of occurences of process τ_{ab} in graph $\mathcal{G}_a$
$m(\tau_{ab}, f)$	The number of occurences of process τ_{ab} in the faultlist, f
$pl(\tau^f_{ij/r})$	The set of processes in graph $\mathcal{G}_i$ which preceeds process $\tau^f_{ij/r}$
NF_{is}	Number of faults occured in scenario s_{is}
$MA(\tau_{ab})$	Set of nodes on which process τ_{ab} can be mapped
$DOS(\mathcal{G}_i)$	Degree of schedulability for graph $\mathcal{G}_i$
$DOS(\mathrm{s}_{is})$	Degree of schedulability for scenario s_{is}
$DP(\mathrm{s}_i)$	The processes in scenario s_{is} that included in the calculation of the degree of schedulability
$SL(\tau^f_{ab/r})$	The set of scenarios which process $\tau^f_{ab/r}$ is a part of
$PX(\tau_{ab})$	The set of processes in the process graph that precedes process τ_{ab} and is protected by reexecution
$NO(\tau_{ab}, \kappa)$	Number of occurences of process τ_{ab} in graph $\mathcal{G}_a$ for κ
$\rho(N_n, \tau^f_{ab/r})$	Utilisation of processing element N_n for the scenarios that process $\tau^f_{ab/r}$ is in
$SU(N_n, \tau^f_{ab/r})$	The set of processes that are mapped on processing element N_n and has at least one scenario s_{is} in common with process $\tau^f_{ab/r}$
$SPS_i(\tau^f_{ab/r})$	The set of processes from a given graph $\mathcal{G}_i$ that process $\tau^f_{ab/r}$ can co-exist with

$ISS(\tau^{f_1}_{ab/r_1}, \tau^{f_2}_{ij/r_2})$	Whether two processes can occur together. They must have at least one scenario in common
$R^{w}_{ab(f)(r)}$	Worst case response time of process $\tau^{f}_{ab/r}$
$C^{b}_{ab(f)(r)}$	Best case response time of process $\tau^{f}_{ab/r}$
$\mathcal{M}(\tau^{f}_{ab/r})$	The mapping of process $\tau^{f}_{ab/r}$
${P_{ab/r}}^{f}$	Priority of process $\tau^{f}_{ab/r}$
${P_{ab/r}}^{m,f}$	Priority of process $\tau^{f}_{ab/r}$
NOP_{Γ_i}	Number of processes in transaction Γ_i
$NOA_{\mathcal{G}_i}$	Number of processes and messages in graph $\mathcal{G}_i$

Data Structures

LPFL	Given a process and number of faults it returns the corresponding list of processes
LFTA	Given a process and a faultlist, it returns the corresponding process and its replicas

Appendix B

List of Abbreviations

BCET	Best Case Execution Time
BCRT	Best Case Response Time
BCTT	Best Case Transmission Time
CAN	Controller Area Network
DOS	Degree of Schedulability
FTPG	Fault-Tolerant Process Graph
ISS	In Same Scenario
PE	Processing Element
PG	Process Graph
RTA	Response Time Analysis
WCET	Worst Case Execution Time
WCRT	Worst Case Response Time
WCTT	Worst Case Transmission Time

Appendix C

Pseudocode

This appendix contains the pseudocode of the algorithms describe in this thesis.

Notice that when using the function *copy()* to copy a process or a message, only the properties of the message is copied, such as mapping table, current mapping and priority. Any graph related information is not copied.

The function *connect*(τ_a, τ_b) connects processes mutually such that process τ_b becomes a successor to τ_a and process τ_a becomes a predecessor of τ_b.

In the pseudocode, we will use the notation A_{ab}, which can be either a process (τ_{ab}) or a message (m_{ab}). For the FTPG $A^{f}_{ab/r}$ is likewise either a process (τ_{ab}fr) or a message ($m_{ab(f)(r)}$).

Algorithm 4 Modified *BranchInterference(τ_{ab}, τ_{ik}, τ_{iB}, w, p)*

```
SB=$succ(\tau_{iB})$
**if** $\tau_{iB} \in hp_i(\tau_{ab})$ **then**
   $SB = SB \cup \tau_{iB}$
**end if**
**for** $\tau_{im} \in SB$ such that $\tau_{im} \in hp_i(\tau_{ab})$ **do**
   $S = \{\tau_{il} \in H_{im}(\tau_{ab}) \mid \tau_{iB} < \tau_{il}\}$
   $sectionIS = \emptyset$
   **for** $\tau_{ij} \in S$ such that $TaskInterference(\tau_{ab}, \tau_{ik}, \tau_{ij}, w, t) > 0$ **do**
      $sectionIS = sectionIS \cup \tau_{ij}$
   **end for**
   $SB = SB \cup succ(H^{seg}_{im}(\tau_{ab})) - H^{seg}_{im}(\tau_{ab})$
**end for**
**for all** $\tau_{is} \in SB$ **do**
   $[bIS, bD] = BranchInterference(\tau_{ab}, \tau_{ik}, \tau_{is}, w, p)$
   $subBranchesIS = subBranchesIS \cup bIS$
   $subBDelta = \max(subBDelta, bD)$
**end for**
**if** ($\tau_{iB} \in pred(H^{seg}_{im}(\tau_{ab}))$ and $H^{seg}_{im}(\tau_{ab})$ is blocking {see Eq. (3.35)}) **then**
   $branchIS = subBranchesIS$
   $branchDelta = \max(\mathbb{C}(sectionIS) - \mathbb{C}(subBranchesIS), subBDelta)$
**else**
   $branchIS = \max_{\mathbb{C}}(sectionIS, subBranchesIS)$
   $branchDelta = \max(\mathbb{C}(subBranchesIS) + subBDelta - \mathbb{C}(branchIS), 0)$
**end if**
**return** [$branchIS$,$branchDelta$]
```

Algorithm 5 splitIntoScenarios($\mathcal{G}_a$)

```
Require: $\mathcal{G}_a$ must be a valid Fault-Tolerant Process Graph
Require: The $\Gamma_a$ corresponding to $\mathcal{G}_a$ can only have one sink node
  $\mathcal{G}_b \Leftarrow$ Initialize new empty transaction
  $mapOldWithNew \Leftarrow$ Initialize new hashmap
  for all $\tau^f_{ab/r} \in \mathcal{G}_a$ do
    if $succ(\tau^f_{ab/r}) = \emptyset$ then
      $\tau^f_{bb/r} \Leftarrow$ copy($\tau^f_{ab/r}$)
      add $\tau^f_{bb/r}$ to $\mathcal{G}_b$
      add ($\tau^f_{ab/r}$, $\tau^f_{bb/r}$) to mapOldWithNew
      for all $\tau^f_{ap/r} \in pred(\tau^f_{ab/r})$ do
        splitRecursively($\tau^f_{bb/r}$, $\tau^f_{ap/r}$, $\mathcal{G}_b$, mapOldWithNew)
      end for
    end if
  end for
```

Algorithm 6 splitRecursively($A^f_{bb/r}$, $A^f_{ac/r}$, $\mathcal{G}_b$, mapOldWithNew)

```
  if $A^f_{ac/r} \notin mapOldWithNew$ then
    $A^f_{bc/r} \Leftarrow$ copy($A^f_{ac/r}$)
    add ($A^f_{ac/r}$, $A^f_{bc/r}$) to mapOldWithNew
    connect($A^f_{bc/r}$, $A^f_{bb/r}$)
    for all $A^f_{ap/r} \in pred(A^f_{ab/r})$ do
      splitRecursively($A^f_{bc/r}$, $A^f_{ap/r}$, $\mathcal{G}_b$, mapOldWithNew)
    end for
  else
    $A^f_{bc/r} \Leftarrow$ lookup $A^f_{ac/r}$ in mapOldWithNew
    connect($A^f_{bb/r}$, $A^f_{bc/r}$)
  end if
```

Algorithm 7 addReexecution(τ_{ab})

```
Require: $\tau_{ab}$: the original process to add reexecution
  for $i = 0 \ldots \kappa - 1$ do
    res ⇐ LPFL.lookup($\tau_{ab}$, i)
    for all $\tau^{f}_{ab/r} \in res$ do
      $\tau^{f_2}_{ab/r}$ ⇐ copy($\tau^{f}_{ab/r}$)
      let $f_2 = f_b \uplus \tau_{ab}$
      connect($\tau^{f}_{ab/r}$, $\tau^{f_2}_{ab/r}$)
      mark transition from $\tau^{f}_{ab/r}$ to $\tau^{f_2}_{ab/r}$ with fault condition
      listToAddReplication ⇐ Initialize new empty list
      buildReexecutionRecursively($\tau^{f}_{ab/r}$, $\tau_{ab}$, listToAddReplication)
      for all $\tau^{f}_{ar/r} \in listToAddReplication$ do
        addReplicationAux($\tau^{f}_{ar/r}$)
      end for
    end for
  end for
```

Algorithm 8 buildReexecutionRecursively($A^{f}_{ab/r}$, A_{ab}, listToAddReplication)

```
if $A_{ab} \in \mathcal{P}_x$ and $|f| < \kappa$ then
  $A^{f_2}_{ab/r} \Leftarrow$ copy($A^{f}_{ab/r}$)
  let $f_2 = f_b \uplus \tau_b$
  LFTA.put($A^{f_2}_{ab}$), LPFL[$A_{ab}$, $|f_2|$].insert($A^{f_2}_{ab/r}$)
  connect($A^{f}_{ab/r}$, $A^{f_2}_{ab/r}$)
  mark transition from $A^{f}_{ab/r}$ to $A^{f_2}_{ab/r}$ with fault condition
  buildReexecutionRecursively($A^{f_2}_{ab/r}$, $A_{ab}$, listToAddReplication)
end if
for $A_{ac} \in succOrg(A_{ab})$ do
  if $|predOrg(A_{ac})| = 1$ then
    $A^{f_c}_{ac/r_c} \Leftarrow$ copy($A_{ac}$)
    let $f_c = f, r_s = 0$
    LFTA.put($A^{f_c}_{ac}$), LPFL[$A_{ac}$, $|f_2|$].insert($A^{f_c}_{ac/r_c}$)
    if ($\tau_{ac}$ is process) then
      if $\tau_{ac} \in \mathcal{P}_r$ and $|f_s| < \kappa$ then
        add $A^{f_c}_{ac/r_c}$ to listToAddReplication
      end if
      for all $A^{f_p}_{ap/r_p} \in LFTA.lookup(A^{f}_{ab/r})$ do
        connect($A^{f_p}_{ap/r_p}$, $A^{f_c}_{ac/r_c}$)
      end for
    else
      connect($A^{f}_{ab/r}$, $A^{f_s}_{as/r_s}$)
    end if
    buildReexecutionRecursively($A^{f_2}_{ab/r}$, $A_{ab}$, listToAddReplication)
  else
    permutations $\Leftarrow$ createAllAllowedPermutations($A_{ac}$, f)
    for all $permutation \in permutations$ do
      if ($LFTA.lookup(permutation) = \emptyset$) then
        $A^{f_p}_{ac/r_p} \Leftarrow$ copy($A_{ac}$)
        $f_p$ = faultlist from permutation
        LFTA.put($A^{f_p}_{ac}$), LPFL[$A_{ac}$, $|f_p|$].insert($A^{f_p}_{ac/r_p}$)
        for all $A^{f_o}_{ao/r_o} \in permutation$ do
          connect($A^{f_o}_{ao/r_o}$, $A^{f_p}_{ac/r_p}$)
        end for
      end if
    end for
  end if
end for
```

Algorithm 9 createAllAllowedPermutations(τ_{ab}, f)

```
Require: $\tau_{ab}$ : process of which predecessors are to be permutated
Require: $f$ : The minimum faultlist of the process to be permutated
  pL ⇐ Initialise empty list
  init ⇐ true
  for all $pred \in predOrg(\tau_{ab})$ do
    if init then
      for $i = 0 \ldots \kappa$ do
        curList = LPFL[$A_{ab}$, i]
        for all $cur \in curList$ do
          add cur to pL
        end for
      end for
      init ⇐ false
    else
      for $i = 0 \ldots \kappa$ do
        curList = LPFL[$A_{ab}$, i]
        for all $cur \in curList$ do
          for all $cand \in pL$ do
            remove cand from pL
            if $cur \cup cand$ is allowed permutation {acc. to subsection 4.4.1}
            then
              add $cur \cup cand$ to pL
            end if
          end for
        end for
      end for
    end if
  end for
  return pL
```

Algorithm 10 addReplication(τ_{ab})

```
  for $i = 0 \ldots \kappa - 1$ do
    res ⇐ LPFL[$\tau_{ab}$, i]
    for all $\tau_{ab}^{f} \in res$ do
      addReplicationAux($\tau_{ab}^{f}$)
    end for
  end for
```

Algorithm 11 addReplicationAux(τ_{ab}^{f})

```
Require: $\tau_{ab}^{f}$: Process to add replication
  for $i = 1 \ldots (k - |f|)$ do
    $\tau_{ab/i}^{f} \Leftarrow$ copy($\tau_{ab}^{f}$)
    LFTA($\tau_{ab}^{f}$).put($\tau_{ab/i}^{f}$)
    for all $A_{ap/r_p}^{f_p} \in pred(\tau_{ab}^{f})$ do
      connect $A_{ap/r_p}^{f_p}$, $\tau_{ab/i}^{f}$
    end for
    for all $A_{as/r_s}^{f_s} \in succ(\tau_{ab/0}^{f})$ do
      if $A_{as/r_s}^{f_s}$ is message then
        $m_{as/i}^{f_s} \Leftarrow$ copy($m_{as/r_s}^{f_s}$)
        LFTA($m_{as}^{f_s}$).put($m_{as/i}^{f_s}$)
        connect $\tau_{ab/i}^{f}$, $m_{as/i}^{f_s}$
        for $A_{at/r_t}^{f_t}$ in $succ(A_{as/r_s}^{f_s})$ do
          connect $m_{as/i}^{f_s}$, $A_{at/r_t}^{f_t}$
        end for
      else
        connect $\tau_{ab/i}^{f}$, $A_{as/r_s}^{f_s}$
      end if
    end for
    add to tables
  end for
```

Algorithm 12 removeRexecution(τ_{ab})

```
  for $i = 0 \ldots k - 1$ do
    for all $\tau_{ab}^{f} \in LPFL.lookup(\tau_{ab}, i)$ do
      if $\tau_{ab} \in f$ then
        removeSuccAux($\tau_{ab}^{f}$)
      end if
    end for
  end for
```

Algorithm 13 removeSuccAux(τ_{ab}^{f})

```
Require: $\tau_{ab/r}^{f}$ : to be removed
  remove $\tau_{ab}^{f}$ from graph
  remove $\tau_{ab}^{f}$ from LFTA
  remove $\tau_{ab}^{f}$ from LPFL[$\tau_{ab}$,$|f|$]
  for all $\tau_{ac/r_c}^{f_c} \in succ(\tau_{ab/r}^{f})$ do
    removeSuccAux($\tau_{ac/r_c}^{f_c}$)
  end for
```

Algorithm 14 removeReplication(τ_{ab})

```
Require: $\tau_{ab}$: The process to have replication removed
  for $i = 0 \dots k-1$ do
    for all $\tau_{ab}^{f} \in LPFL[\tau_{ab}, i]$ do
      pList = LFTA($\tau_{ab}^{f}$)
      for $j = 1 \dots |pList| - 1$ do
        remove all references to pList[i] among its successors and predecessors

        remove pList[i] from LFTA($\tau_{ab}^{f}$)
      end for
    end for
  end for
```

Algorithm 15 Generate Random Graphs With The Same Number Of Processes

```
Require: rP: requested number of procs
Require: aP: actual number of procs
Require: nG: number of graphs to generate
Require: sG: set of generated graphs
  $\Gamma_a$ = generateGraph(rP)
  sG $\Leftarrow$Initialize empty set
  sG.add($\Gamma_a$)
  aP = size($\Gamma_a$)
  while size(sG) ¡ nG do
    $\Gamma_b$ = generateGraph using TGFF with rP processes
    if size($\Gamma_b$) == aP then
      sG.add($\Gamma_b$)
    end if
  end while
  return sG
```

Appendix D

Cruise Controller Example

This appendix contains the results of the design optimisation used with the model of adaptive cruise controller. We have included the XML file describing the model and the result produced by the program.

D.1 Input File for Adaptive Cruise Controller Example

```
<?xml version="1.0" encoding="UTF-8"?>

<bench>

   <faultmodel numberOfFaults="2"/>

   <platform>

      <nodes>
         <node number="1"/>
         <node number="2"/>
         <node number="3"/>
         <node number="4"/>
      </nodes>

      <communicationchannels>
         <channel number="0"/>
      </communicationchannels>

   </platform>

   <application>
      <transactions>
         <transaction number="0" period="1000">
            <!-- Dummy root process -->
```

```
        <process number="0" node="1" priority="1">
          <mappingtable>
            <mapping node="1" wcet="0"/>
            <mapping node="2" wcet="0"/>
            <mapping node="3" wcet="0"/>
            <mapping node="4" wcet="0"/>
          </mappingtable>
        </process>
        <!--Measures the velocity of the vehicle -->
        <process number="1" node="1" priority="1">
          <mappingtable>
            <mapping node="1" bcet="2" wcet="2"/>
          </mappingtable>
          <dependencies>
            <dependency from="0"/>
          </dependencies>
        </process>
        <!-- Measures the distance to the closest vehicle in front -->
        <process number="2" node="1" priority="2">
          <mappingtable>
            <mapping node="1" bcet="2" wcet="2"/>
          </mappingtable>
          <dependencies>
            <dependency from="0"/>
          </dependencies>
        </process>
        <!-- Calculates the relative speed to the vehicle in front -->
        <process number="3" node="2" priority="1">
          <mappingtable>
            <mapping node="1" bcet="2" wcet="6"/>
            <mapping node="2" bcet="2" wcet="6"/>
            <mapping node="3" bcet="2" wcet="6"/>
            <mapping node="4" bcet="2" wcet="6"/>
          </mappingtable>
          <dependencies>
            <dependency from="1"/>
            <dependency from="2"/>
          </dependencies>
        </process>
        <!--Calculates the desired velocity -->
        <process number="4" node="2" priority="1">
          <mappingtable>
            <mapping node="1" bcet="2" wcet="2"/>
            <mapping node="2" bcet="2" wcet="2"/>
            <mapping node="3" bcet="2" wcet="2"/>
            <mapping node="4" bcet="2" wcet="2"/>
          </mappingtable>
          <dependencies>
            <dependency from="3"/>
          </dependencies>
        </process>
        <!--Controls the velocity of the vehicle, indirectly, by changing the
            throttle -->
        <process number="5" node="3" priority="1">
          <mappingtable>
            <mapping node="1" bcet="2" wcet="6"/>
            <mapping node="2" bcet="2" wcet="6"/>
            <mapping node="3" bcet="2" wcet="6"/>
            <mapping node="4" bcet="2" wcet="6"/>
          </mappingtable>
          <dependencies>
            <dependency from="4"/>
          </dependencies>
        </process>
        <!--Controls the physical changing of the throttle -->
        <process number="6" node="4" priority="1">
          <mappingtable>
            <mapping node="4" bcet="2" wcet="2"/>
          </mappingtable>
          <dependencies>
            <dependency from="5"/>
          </dependencies>
        </process>
      </transaction>
    </transactions>
  </application>

</bench>
```

D.2 Results from the Heuristics

The adaptive cruise controller is optimised using our implementation as follows:

```
$ doftes.sh -mode optimize -optimizationMaxIterations 20
                                                   tests/acc.xml
```

And the output is as follows:

```
Optimal configuration found after 4 iterations
Initial degree of schedulability: -2658
Final degree of schedulability: -3115
The solution is schedulable!

Policy Assignment:
        Reexecution: P1, P2, P6
        Replication: P0, P3, P4, P5

Priority and mappings:

        Process Faultlist                 ReplicaNb       Priority        Processing
                                                                          Element
        ----------------------------------------------------------------------------
        P0       []                       0               1               N1
        P0       []                       1               1               N3
        P0       []                       2               1               N2
        P1       [P1]                     0               1               N1
        P1       [P1,P1]                  0               1               N1
        P1       []                       0               1               N1
        P2       [P2]                     0               2               N1
        P2       []                       0               2               N1
        P2       [P2,P2]                  0               2               N1
        P3       [P1]                     0               1               N2
        P3       []                       2               1               N3
        P3       []                       1               1               N1
        P3       [P2]                     1               1               N4
        P3       []                       0               1               N2
        P3       [P2,P2]                  0               1               N2
        P3       [P1]                     1               1               N1
        P3       [P2]                     0               1               N2
        P3       [P1,P2]                  0               1               N2
        P3       [P1,P1]                  0               1               N2
        P4       [P1,P2]                  0               1               N2
        P4       [P1]                     1               1               N1
        P4       [P2,P2]                  0               1               N2
        P4       [P2]                     1               1               N4
        P4       [P1,P1]                  0               1               N2
        P4       []                       0               1               N2
        P4       []                       1               1               N1
        P4       [P1]                     0               1               N2
        P4       [P2]                     0               1               N2
        P4       []                       2               1               N4
        P5       [P2,P2]                  0               1               N3
        P5       [P1,P2]                  0               1               N3
        P5       []                       2               1               N4
        P5       []                       0               1               N3
        P5       []                       1               1               N1
        P5       [P1,P1]                  0               1               N3
        P5       [P2]                     0               1               N3
        P5       [P2]                     1               1               N4
        P5       [P1]                     1               1               N1
        P5       [P1]                     0               1               N3
        P6       [P6]                     0               1               N4
        P6       [P2]                     0               1               N4
        P6       [P2,P6]                  0               1               N4
        P6       [P1,P2]                  0               1               N4
        P6       []                       0               1               N4
        P6       [P1,P6]                  0               1               N4
        P6       [P1]                     0               1               N4
        P6       [P1,P1]                  0               1               N4
        P6       [P6,P6]                  0               1               N4
        P6       [P2,P2]                  0               1               N4
```

Appendix E

Other Examples

E.1 Splitting FTPG into Scenarios

This section contains two examples illustrating how faults scenarios are extracted from complete fault-tolerant process graphs. They all are created by our program, which uses Graphviz[37] as backend renderer. Due to charset limitations the generated graphs have different notations. The processes are represented as ovals with a signature inside. The first line of the signature contains process number denoted with P and processing element in parentheses. The second line contains the fault list and the replica number. The third line is the scenario list, i.e. alls scenarios a process belongs to. Notice that scenario list is empty when the scenarios are splitted. The fault conditions are drawn with red arrows.

E.1.1 Example 1: Reexecution Only

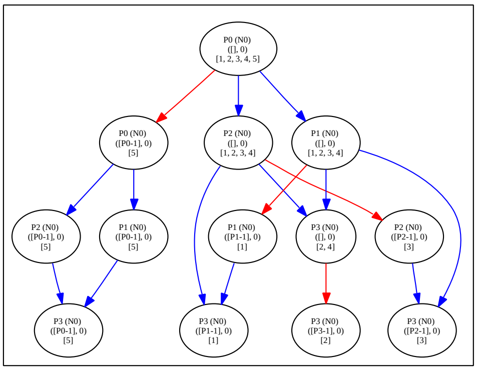
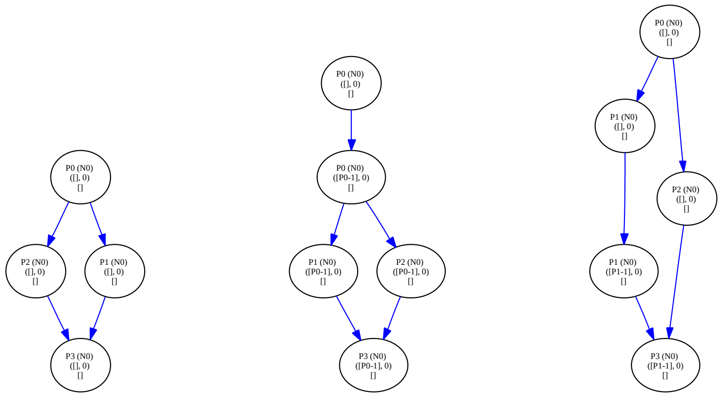


Figure E.1: A example of a diamond-shaped fault-tolerant process graph. All processes are protected by reexecution and $\kappa = 1$

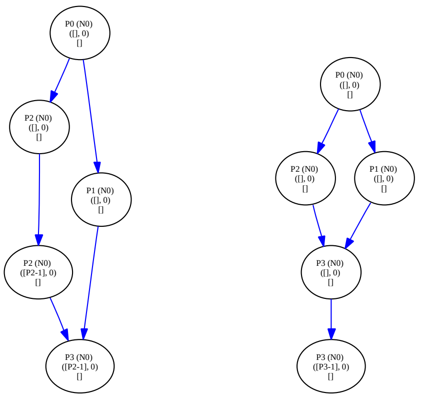


(a) Scenario 1, no faults

(b) Scenario 2, process τ_0 fails

(c) Scenario 3, process τ_1 fails

Figure E.2: Scenarios derived from the graph in Figure E.1



(a) Scenario 4, process τ_2 fails

(b) Scenario 5, process τ_3 fails

Figure E.3: Scenarios derived from the graph in Figure E.1(continued)

E.1.2 Example 2: Reexecution Combined With Replication

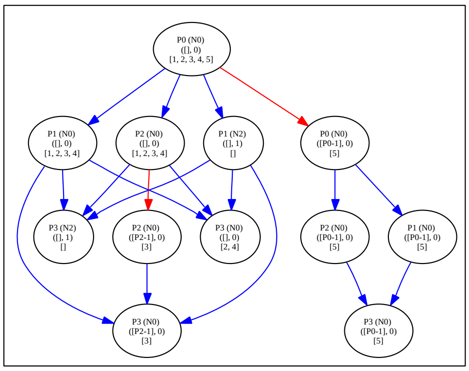
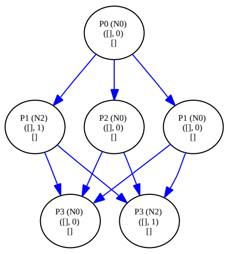
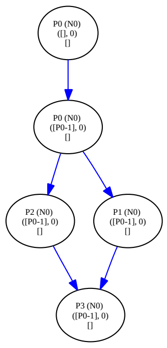


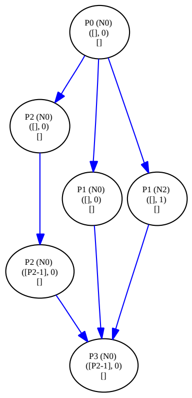
Figure E.4: A example of a diamond-shaped fault-tolerant process graph. Processes τ_0 and τ_2 are protected with reexecution, whereas τ_1 and τ_3 are replicated, $\kappa = 1$



(a) Scenario 1, no faults



(b) Scenario 2, process τ_0 fails



(c) Scenario 3, process τ_2 fails

Figure E.5: Scenarios derived from the graph in Figure E.4

E.2 LFTA and LPFL

This section contains illustrations explaining the meaning of the data structures (lookup tables LFTA and LPFL) used in the operations on fault-tolerant process graphs. Then shown tables hold the information about the processes in the graph from Figure E.7.

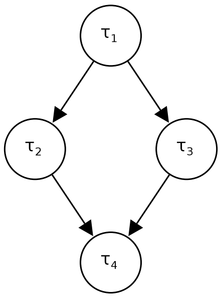


Figure E.6: Initial Process Graph Shaped as a Diamond

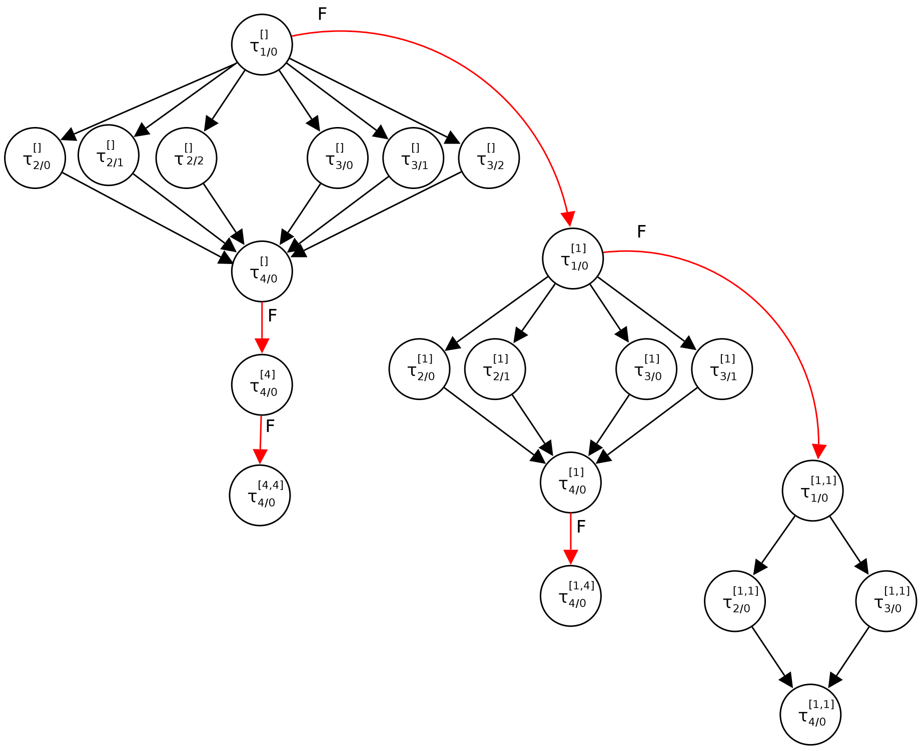


Figure E.7: The figure shows the fault-tolerant application derived from the graph in Figure E.6, where $\kappa = 2$, $(\tau_1, \tau_4) \in \mathcal{P}_x$, $(\tau_2, \tau_3) \in \mathcal{P}_r$

$h(\tau^{f}_{ab/0})$	The corresponding elements
$h(\tau_1^{[\,]})$	$\tau_{1/0}^{[\,]}$
$h(\tau_2^{[\,]})$	$\tau_{2/0}^{[\,]}$, $\tau_{2/1}^{[\,]}$, $\tau_{2/2}^{[\,]}$
$h(\tau_3^{[\,]})$	$\tau_{3/0}^{[\,]}$, $\tau_{3/1}^{[\,]}$, $\tau_{3/2}^{[\,]}$
$h(\tau_4^{[\,]})$	$\tau_{4/0}^{[\,]}$
$h(\tau_4^{[4]})$	$\tau_{4/0}^{[4]}$
$h(\tau_4^{[4,4]})$	$\tau_{4/0}^{[4,4]}$
$h(\tau_1^{[1]})$	$\tau_{1/0}^{[1]}$
$h(\tau_2^{[1]})$	$\tau_{2/0}^{[1]}$, $\tau_{2/1}^{[1]}$
$h(\tau_3^{[1]})$	$\tau_{3/0}^{[1]}$, $\tau_{3/1}^{[1]}$
$h(\tau_4^{[1]})$	$\tau_{4/0}^{[1]}$
$h(\tau_4^{[1,4]})$	$\tau_{4/0}^{[1,4]}$
$h(\tau_1^{[1,1]})$	$\tau_{1/0}^{[1,1]}$
$h(\tau_2^{[1,1]})$	$\tau_{2/0}^{[1,1]}$
$h(\tau_3^{[1,1]})$	$\tau_{3/0}^{[1,1]}$
$h(\tau_4^{[1,1]})$	$\tau_{4/0}^{[1,1]}$

Table E.1: Contents of LFTA for Figure E.7

	$\lvert f_{ab}\rvert$		
	0	1	2
τ_1	$\tau_1^{[\,]}$	$\tau_1^{[1]}$	$\tau_1^{[1,1]}$
τ_2	$\tau_2^{[\,]}$	$\tau_2^{[1]}$	$\tau_2^{[1,1]}$
τ_3	$\tau_3^{[\,]}$	$\tau_3^{[1]}$	$\tau_3^{[1,1]}$
τ_4	$\tau_4^{[\,]}$	$\tau_4^{[1]}$, $\tau_4^{[4]}$	$\tau_4^{[1,1]}$, $\tau_4^{[1,4]}$

Table E.2: Contents of LPFL for Figure E.7

E.3 Calculating Degree of Schedulability

The tables shown in this section illustrate how the degree of schedulability is computed for the exaple from Section E.1.1.

$\tau_{ij/r}^{f}$	P_{ij}	C_{ij}	D_{ij}	s$_l$ 1	2	3	4	5
NF_{s_l}				0	1	1	1	1
$\tau_{0/0}^{[\,]}$	7	2	10	**2**	2	**2**	**2**	**2**
$\tau_{1/0}^{[\,]}$	5	4	30	**6**		6	**6**	**6**
$\tau_{2/0}^{[\,]}$	3	5	30	**7**		**7**	7	**7**
$\tau_{3/0}^{[\,]}$	5	10	40	**17**				17
$\tau_{0/0}^{[0]}$					**4**			
$\tau_{1/0}^{[0]}$					**8**			
$\tau_{2/0}^{[0]}$					**9**			
$\tau_{3/0}^{[0]}$					**19**			
$\tau_{1/0}^{[1]}$						**10**		
$\tau_{2/0}^{[2]}$							**12**	
$\tau_{3/0}^{[3]}$								**27**
$\tau_{3/0}^{[1]}$						**20**		
$\tau_{3/0}^{[2]}$							**22**	
$\sum_{\tau_{ab} \in DP(s_i)} max(0, R_i^w - D_i)$				0	0	0	0	0
$\sum_{\tau_{ab} \in DP(s_i)} \left(R_i^w - D_i\right)$				-78	-70	-71	-68	-68
w_{ij}				1	1/2	1/2	1/2	1/2
$DOS(i)$				-78	-35	-35.5	-34	-34
Total cost	-216.5							
$\lvert\mathcal{S}_j\rvert$	5							
Total cost / $\lvert\mathcal{S}_j\rvert$	-43.3							

Table E.3: Calculation of the degree of schedulability for the example 1 in Section E.1.1.

				s_l		
$\tau^{f}_{ij/r}$	P_{ij}	C_{ij}	D_{ij}	1	2	3
NF_{s_l}				0	1	1
$\tau^{[\,]}_{0/0}$	7	2	10	**2**	2	**2**
$\tau^{[\,]}_{1/0}$	5	4	30	**7**		**6**
$\tau^{[\,]}_{1/1}$	5	4	30	6		**6**
$\tau^{[\,]}_{2/0}$	3	5	30	**7**		7
$\tau^{[\,]}_{3/0}$	5	10	40	**17**		
$\tau^{[\,]}_{3/1}$	5	10	40	17		
$\tau^{[0]}_{0/0}$					**4**	
$\tau^{[0]}_{1/0}$					**8**	
$\tau^{[0]}_{2/0}$					**9**	
$\tau^{[0]}_{3/0}$					**19**	
$\tau^{[2]}_{2/0}$						**12**
$\tau^{[2]}_{3/0}$						**22**
$\sum_{\tau_{ab}\in DP(s_i)} max(0, R^w_i - D_i)$				0	0	0
$\sum_{\tau_{ab}\in DP(s_i)} \left(R^w_i - D_i\right)$				-78	-70	-71
w_{ij}				1	1/2	1/2
$DOS(i)$				-77	-35	-34
Total cost	-147					
$\lvert\mathcal{S}_j\rvert$	3					
Total cost / $\lvert\mathcal{S}_j\rvert$	-49					

Table E.4: Calculation of the degree of schedulability for the example 2 in Section E.1.2.

E.4 Convergence of the Heuristic

This section contains the output from the program illustrating the iteration of the optimisation heuristics.

Listing E.1: Start guess: All processes are protected with reexecution

```
Initial cost: 1

1. Iteration
BEST move: FaultToleranceMove: P0, REEXECUTION -> REPLICATION (Cost: -19)

2. Iteration
BEST move: FaultToleranceMove: P3, REEXECUTION -> REPLICATION (Cost: -21)

3. Iteration
BEST move: FaultToleranceMove: P4, REEXECUTION -> REPLICATION (Cost: -22)

Number of iterations: 3
Final cost: -22
Final csonfiguration:
 P0: REPLICATION
   (P0, [FTA: 0; []], RN=1) P = 4 PE = N2
 P1: REEXECUTION
 P2: REEXECUTION
 P3: REPLICATION
   (P3, [FTA: 3; []], RN=1) P = 1 PE = N1
 P4: REPLICATION
   (P4, [FTA: 4; []], RN=1) P = 6 PE = N2
```

Listing E.2: Start guess: All processes are protected with replication

```
Initial cost: 4

1. Iteration
BEST move: FaultToleranceMove: P2, REPLICATION -> REEXECUTION (Cost: -22)

Number of iterations: 1
Final cost : -22
Final configuration:
 P0: REPLICATION
   (P0, [FTA: 0; []], RN=1) P = 4 PE = N1
 P1: REEXECUTION
 P2: REEXECUTION
 P3: REPLICATION
   (P3, [FTA: 3; []], RN=1) P = 1 PE = N2
 P4: REPLICATION
   (P4, [FTA: 4; []], RN=1) P = 6 PE = N1
```

Appendix F

Program

F.1 Class Diagrams

This section contains four UML class diagrams that illustrate the most important classes and relations, which are necessary to get an overview of how the program is implemented. For this reason the methods are shown without parameters and some secondary attributes are omitted.

Figure F.1 explains the relations between the data structures related to our software model. The hardware model shown in Figure F.2. The implementation of the heuristics can be seen in Figure F.3, and the response time analysis algorithm is composed as shown in Figure F.4.

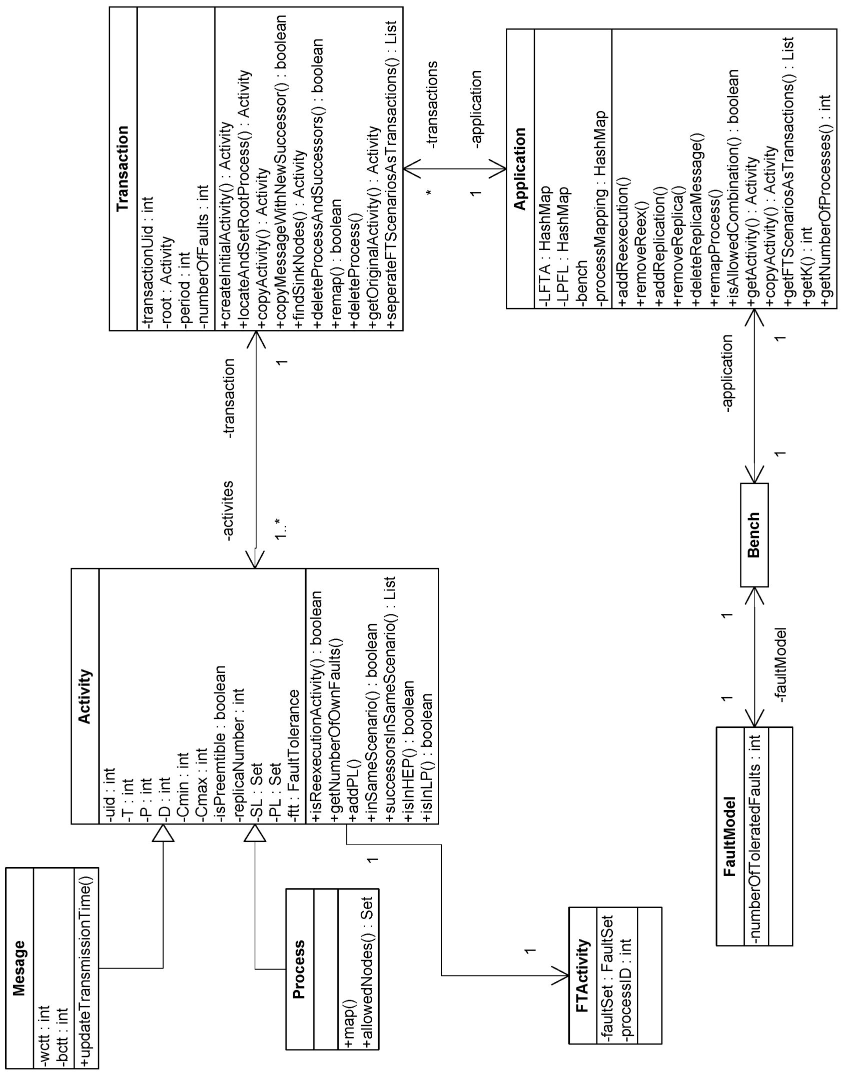


Figure F.1: Class Diagram of the Application Model

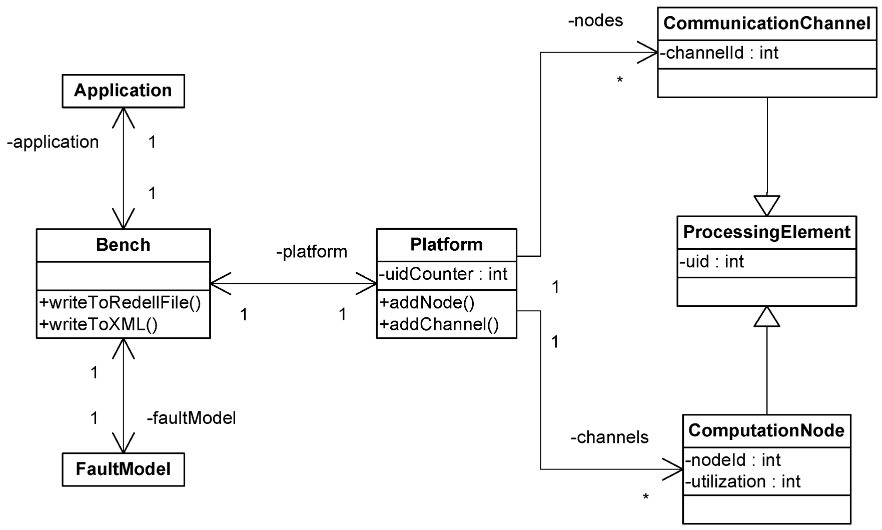


Figure F.2: Class Diagram of the Hardware Model

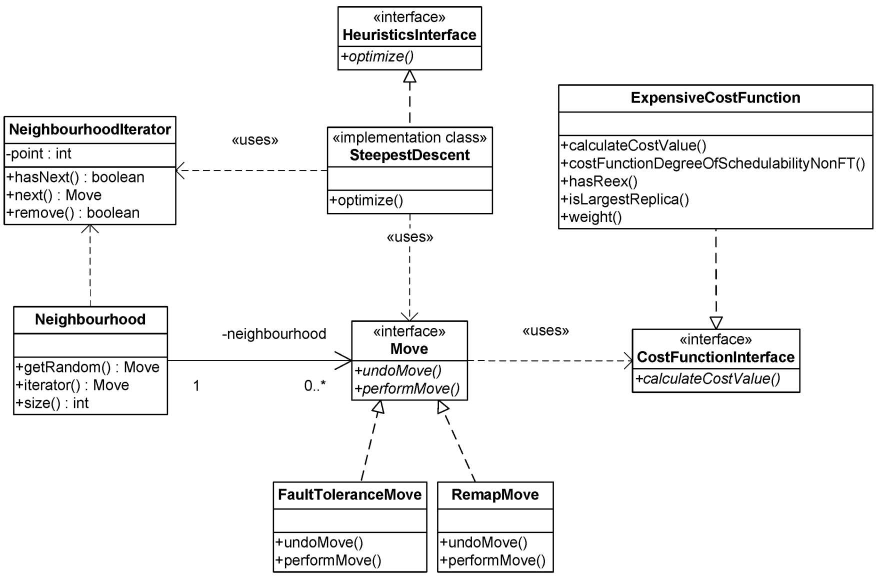


Figure F.3: Class Diagram of the Heuristics

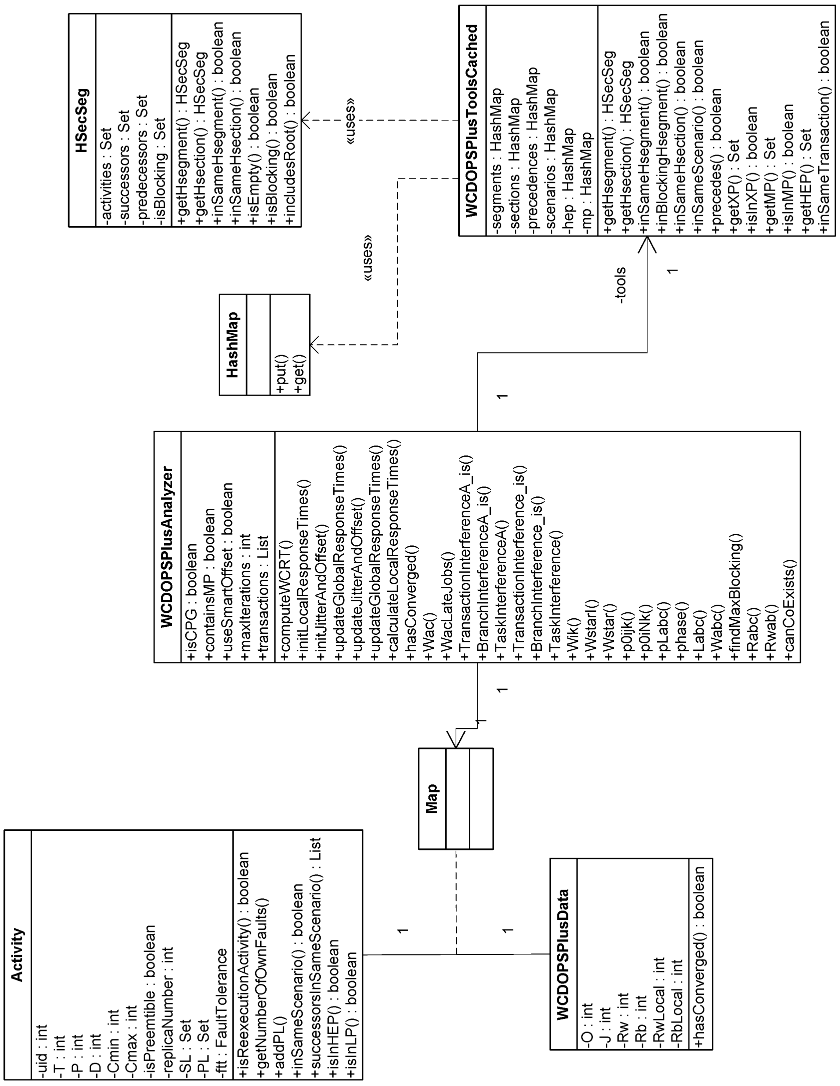


Figure F.4: Class Diagram of the Response Time Analysis

F.2 Command Line Manual

The program can be started either from the Eclipse IDE, when the thesis project is opened. However, we have pre-build the newest version of code, located in folder *doftes.build*. Together with the code, we have placed all necessary libraries and a couple of examples to demonstrate how the program can be used. To simplify the start of the program, we have written a very simple bash script that will start the program. With this script the program can be lauched from the command line as follows:

```
$ doftes.sh [options...]  benchfile
```

The *benchfile* given as an argument is the path to the file containing the bench description of the system. It must always be given, when the user starts the program. The options are listed below:

- `mode` ***STR***: selects the behaviour of the program, can have one of the following values: *show*, *rta*, *optimize* (default). The example will show the application graph from the bench file:
  ```
  $  doftes.sh -mode show tests/diamond1.xml
  ```
- `rtaMaxIterations` ***INT***: maximum number of iterations, when working in *rta* mode. If the analysis has not converged after the given number of iterations, the program stops. Default value is 20. Example with 5 iterations:
  ```
  $  doftes.sh -mode rta -rtaMaxIterations 5 tests/diamond1.xml
  ```
- `rtaDebugLevel` ***INT***: debug output granularity, when working in *rta* mode. Default value is 0 (no debug output). In the example changed to 10:
  ```
  $  doftes.sh -mode rta -rtaDebugLevel 10 tests/diamond1.xml
  ```
- `rtaDebugFile` ***STR***: set the file to which debug output should redirected, in *rta* mode. If the not given, the output is printed to the console stdout. Print debug output to file out.txt with granularity of 15:
  ```
  $  doftes.sh -mode rta -rtaDebugLevel 15 -rtaDebugFile out.txt
  tests/diamond1.xml
  ```
- `rtaDebugMeasureTime`: if present, the debug output will include time measurements for all functions calls. To be used in *rta* mode, when testing and profiling the program:
  ```
  $  doftes.sh -mode rta -rtaDebugMeasureTime tests/diamond1.xml
  ```

- `rtaNaiveOffset`: if present, the smart offset will be disable during the analysis in *rta* mode, for testing purposes:
  ```
  $ doftes.sh -mode rta -rtaNaiveOffset tests/diamond1.xml
  ```
- `rtaDisableBuffer`: if present, disables buffering of segment and sections in *rta* mode, for performance evaluations:
  ```
  $ doftes.sh -mode rta -rtaDisableBuffer tests/diamond1.xml
  ```
- `optimizeMaxIterations` ***INT***: maximum number of iterations, when working in *optimize* mode. If the optimisation has not converged after the given number of iterations, the program stops. Default value is 20. Example with 5 iterations:
  ```
  $ doftes.sh -mode optimize -optimizeMaxIterations 5 tests/diamond1.xml
  ```

F.3 XML Schema for Input Files

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="doftes"
    xmlns="doftes"
    elementFormDefault="qualified">

   <xs:element name="bench" type="BenchType"/>

   <xs:complexType name="BenchType">
      <xs:sequence>
         <xs:element name="faultmodel" type="FaultModelType"/>
         <xs:element name="platform " type="PlatformType"/>
         <xs:element name="application" type="ApplicationType"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="FaultModelType">
      <xs:attribute name="numberOfFaults" type="xs:int"/>
   </xs:complexType>

   <xs:complexType name="PlatformType">
      <xs:all>
         <xs:element name="nodes" type="NodesType"/>
         <xs:element name="communicationchannels" type="CommunicationChannelsType"/>
      </xs:all>
       </xs:complexType>

   <xs:complexType name="NodesType">
      <xs:sequence>
         <xs:element name="node" type="NodeType"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="NodeType">
      <xs:attribute name="number" type="xs:int"/>
   </xs:complexType>

   <xs:complexType name="CommunicationChannelsType">
      <xs:sequence>
         <xs:element name="channel" type="ChannelType"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="ChannelType">
      <xs:attribute name="number" type="xs:int"/>
   </xs:complexType>


   <xs:complexType name="ApplicationType">
      <xs:all>
```

```xml
         <xs:element name="transactions" type="TransactionsType"/>
      </xs:all>
   </xs:complexType>

   <xs:complexType name="TransactionsType">
      <xs:sequence>
         <xs:element name="transaction" type="TransactionType"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="TransactionType">
      <xs:sequence>
         <xs:element name="process" type="ProcessType"/>
      </xs:sequence>
      <xs:attribute name="number" type="xs:int"/>
      <xs:attribute name="period" type="xs:int"/>
   </xs:complexType>

   <xs:complexType name="ProcessType">
      <xs:all>
         <xs:element name="mappingtable" type="MappingtableType"/>
         <xs:element name="dependencies" type="DependenciesType"/>
      </xs:all>
      <xs:attribute name="number" type="xs:int"/>
      <xs:attribute name="node" type="xs:int"/>
      <xs:attribute name="deadline" type="xs:int"/>
      <xs:attribute name="priority" type="xs:int"/>
   </xs:complexType>

   <xs:complexType name="MappingtableType">
      <xs:sequence>
         <xs:element name="mapping" type="MappingType"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="MappingType">
      <xs:attribute name="node" type="xs:int"/>
      <xs:attribute name="bcet" type="xs:int"/>
      <xs:attribute name="wcet" type="xs:int"/>
   </xs:complexType>

   <xs:complexType name="DependenciesType">
      <xs:sequence>
         <xs:element name="dependency" type="DependencyType"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="DependencyType">
      <xs:all>
         <xs:element name="message" type="MessageType"/>
      </xs:all>
      <xs:attribute name="from" type="xs:int"/>
   </xs:complexType>

   <xs:complexType name="MessageType">
      <xs:attribute name="channel" type="xs:int"/>
      <xs:attribute name="priority" type="xs:int"/>
      <xs:attribute name="deadline" type="xs:int"/>
      <xs:attribute name="bctt" type="xs:int"/>
      <xs:attribute name="wctt" type="xs:int"/>
   </xs:complexType>

</xs:schema>
```

Appendix G

Testing

G.1 Sanity Checks for Fault Tolerant Conditional Process Graphs

Another type of unit tests conducted on the graph operations is done by generating random graphs and applying a series of random modifications to the graphs. For all elements in the graph the following sanity requirements are tested:

- **Predecessor Check** ensures that all each process has the correct predecessors, test-case `checkCorrectPredessors`.
- **Fault List Check** tests, whether a process is connected to through the correct combination of fault conditions, test-case `checkCorrectFTA`.
- **Message Sender Check** is used to test that all messages have only one sending process, test-case `checkOnlyOneSenderPerMessage`.
- **Graph Connectivity Check** verifies that all links between nodes are bidirectional, test-case `checkSuccessorEqualsPredecessor`.
- **Fault Scenario Check** to figure out, whether all expected permutations (the occurrences in fault scenarios) of a given process exist, test-case `checkPermutations`.

- **Message Mapping Check** tests, whether all instances of a given message are mapped on the same communication channel as the original message, test-case `checkMessagesSameCommunicationChannel`.
- **Process Replica Check** guarantees that all replicas of a given process is mapped on different processing elements, test-case `checkReplicaOnDifferentPEs`.

All of the checks above are implemented in *structures.app.Transaction*.

G.2 Input for TGFF

```
# Uses "Series-parallel commands (new algorithm)"
gen_series_parallel true

seed 1

# Sets the width of series chains (average, multiplier). This is the number of
    parallel chains generated for each node that is the head of a set of chains
series_wid 2 1

# Sets the length of series chains (average, multiplier).
series_len 2 1

# This allows chains to not rejoin with probability .7
series_subgraph_fork_out .0
series_local_xover 2

# How many graphs to generate
tg_cnt 1;
# Sets the minimum number of tasks per task graph (average, multiplier)
task_cnt 10 1
# Sets the probability that a graph has more than one start node (default 0.0)
prob_multi_start_nodes 0.0
# Sets the laxity of periods relative to deadlines (default 1). Indicates whether
    task graphs deadlines are greater than, less than, or equal to the periods
period_laxity 1
# Sets the multipliers for periods in multirate systems. Allows the user to specify
    the periods relative to each other. The multipliers are randomly selected from
    this list
period_mul 1, 1, 1
# If set, force subgraphs formed in series chains to rejoin into the main graph
series_must_rejoin true
#Sets the probability that a deadline will be hard (vs. soft)
prob_hard_deadline 1.0
# Sets the average time per task including communication. This value is used in
    setting deadlines
task_trans_time 100
# If true, then periods values are forced to be greater than deadlines (default true
    ).
period_g_deadline true
# If true, tasks types are forced to be unique (false by default).
task_unique true
task_attrib wcet 100 25 1.0, bcet 50 10 1.0, priority 50 50 1.0, deadline 100 25 1.0
# Write the task graphs [to .tgff file].
tg_write
pe_write
trans_write
# Sets the label used for the current table.
table_label COMMUN
# Sets the number of tables (of current table type) generated
table_cnt 1
# Attribute on the table (one instance for each table)
# table_attrib price 80 20
# Attribe on the type (one instance for each type/line)
type_attrib bctt 50 20 0.5 1.0, wctt 100 20 0.5 1.0, priority 50 50 0.5 1.0,
    deadline 600 600 0.5 1.0
# Write transmission event information [to .tgff file].
trans_write
```

Bibliography

[1] A. Appel, D. August, D. Clark, D. Walker, and M. Martonosi. Project zap, nsf cyber trust meeting poster. `http://www.cs.princeton.edu/sip/projects/zap/cybertrust-poster.pdf`.

[2] A. Bondavalli, F. Di Giandomenico, M. Pizza, and L. Strigini. Optimal discrimination between transient and permanent faults. *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231)*, pages 214–23, 1998.

[3] E. K. Burke and G. Kendall. *Search Science+Business Media, Inc.* Springer, 2005.

[4] A. Burns, K. Tindell, and A. Wellings. Allocating hard real time tasks (an np-hard problem made easy). *RTSYSTS: Real-Time Systems*, 1992.

[5] G. C. Buttazzo. *Hard Real-Time Computing Systems.* Springer, second edition edition, 2004.

[6] L. Carley. Brake-By-Wire. `http://www.aa1car.com/library/2004/bf110412.htm`, 2004.

[7] V. Claesson, S. Poledna, and J. Soderberg. The xbw model for dependable real-time systems. *Parallel and Distributed Systems, 1998. Proceedings., 1998 International Conference on*, pages 130–138, 1998.

[8] J. Clark and K. Tindell. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, 1994.

[9] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.

[10] F. Corno, F. Esposito, M. S. Reorda, and S. Tosato. Evaluating the effects of transient faults on vehicle dynamic performance in automotive systems. *Proceedings. International Test Conference 2004 (IEEE Cat. No.04CH37586)*, pages 1332–9, 2004.

[11] J. Cosgrove and B. Donnelly. Intelligent automotive networks for distributed real-time control. `http://www.irishscientist.ie/2003/contents.asp?contentxml=03p80a.xml&contentxsl=is03pages.xsl`, 2003.

[12] R. Dick, D. Rhodes, and K. Vallerio. Task Graphs For Free (TGFF), v. 3.3. `http://ziyang.ece.northwestern.edu/tgff/`.

[13] A. Doboli, P. Eles, K. Kuchcinski, Z. Peng, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. *Design, Automation and Test in Europe, 1998., Proceedings*, pages 132–138, 1998.

[14] P. Eles, V. Izosimov, Z. Peng, and P. Pop. Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems. *Design, Automation and Test in Europe, 2005. Proceedings*, pages 864–869, 2005.

[15] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. *Design, Automation and Test in Europe, 1998., Proceedings*, pages 132–138, 1998.

[16] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 152–161, 1995.

[17] J. J. G. Garcia, J. C. P. Gutierrez, and M. G. Harbour. Schedulability analysis of distributed hard real-time systems with multiple-event synchronization. *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 15–24, 2000.

[18] J. J. G. Garcia and M. G. Harbour. Optimized priority assignment for tasks and messages in distributed hard real-time systems. *Parallel and Distributed Real-Time Systems, 1995. Proceedings of the Third Workshop on*, pages 124–132, 1995.

[19] M. G. Harbour and J. C. Palencia. Schedulability analysis for tasks with static and dynamic offsets. *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 26–37, 1998.

[20] M. G. Harbour and J. C. Palencia. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 328–339, 1999.

[21] J. P. Hayes, N. Kandasamy, and B. T. Murray. Transparent recovery from intermittent faults in time-triggered distributed systems. *IEEE Transactions on Computers*, 52(2):113–125, 2003.

[22] M. J. Irwin, M. Kandemir, L. Li, N. Vijaykrishnan, and Y. Xie. Reliability-aware co-synthesis for embedded systems. *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*, pages 41–50, 2004.

[23] V. Izosimov. *Scheduling and Optimization of Fault-Tolerant Embedded Systems.* PhD thesis, Linköping Studies in Science and Technology, 2006.

[24] S. Klaus and S.A. Huss. Interrelation of specification method and scheduling results in embedded system design, 2001.

[25] V. Koltun. Discrete structures, lecture notes. `http://www.stanford.edu/class/cs103x/cs103x-notes.pdf`, 2007.

[26] H Kopetz. *Real-time systems: design principles for distributed embedded applications.* Kluwer Academic Publishers, 1997.

[27] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: the mars approach. *IEEE Micro*, 9(1):25–40, 1989.

[28] Incorporated Object Mentor. JUnit. `http://junit.org`.

[29] C. Pinello, L.P. Carloni, and A.L. Sangiovanni-Vincentelli. Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications. *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, 2:1164–1169, 2004.

[30] P. Pop. Slides, DTU course "02229 Safety-Critical Embedded Systems".

[31] P. Pop. *Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems.* PhD thesis, Linköping Studies in Science and Technology, 581 83 Linköping, Sweden, 2003.

[32] K. H. Poulsen. Reliability-aware energy optimization for fault-tolerant embedded MP-SoCs. Master's thesis, Technical University of Denmark, March 2007.

[33] O. Redell. Implementation of "Accounting for precedence constraints in the analysis of tree-shaped transactions in distributed real time systems". `http://www.md.kth.se/~ola/aida/aidalyze1_0.zip`.

[34] O. Redell. Accounting for precedence constraints in the analysis of tree-shaped transactions in distributed real time systems. Technical report TRITA-MMK 2003:4, Department of Machine Design, The Royal Institute of Technology (KTH), 2003. ISRN: KTH/MMK/R–03/4–SE.

[35] O. Redell. Analysis of tree-shaped transactions in distributed real time systems. *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 239–248, 2004.

[36] C. R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* Blackwell Scientific Publications, 1993.

[37] AT&T Research. Graphviz. `http://graphviz.org`.

[38] P. Riis. Simulation of a distributed implementation of an adaptive cruise controller. Master's thesis, Linköping University, June 2007.

[39] K. Tindell. Adding time-offsets to schedulability analysis. *Tchnical Report YCS 221, Dept. of Computer Science, University of York, England*, 1994.

[40] W. Wolf. *Computers as Components: Principles of Embedded Computing System Design.* Morgan Kaufmann Publishers Inc, US, 2005.

[41] T. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1125–1136, 1998.