Reliability Modelling and Analysis of Real-Time Systems

Colin Stanley Perkins

Submission in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

University of York
Department of Electronics

September 1996

Abstract

The reliability modelling and analysis of real-time, fault-tolerant, embedded systems is considered. It is shown that many existing reliability modelling techniques are inadequate for this task, since they model only the overall system reliability, whilst the timing properties of the system are either neglected, or reduced to simple metrics. A new reliability model is derived, which permits the modelling of both overall system reliability, and the probability distribution of system completion and failure times. This model is based on a set of high level system attributes, which it is expected may be estimated from experimental data. The model is applied to the study of recovery block systems, and it is shown that the results obtained are compatible with, and extend, a number of other system reliability models. The thesis concludes with a discussion of the application of more detailed timing information to the scheduling of safety-critical real-time systems. It is shown that the additional timing information available with models such as that developed herein, allows designers to make more informed choices regarding the tradeoff between safety and performance.

Contents

List of Tables

List of Figures

Acknowledgements

This work was supported by the UK Engineering and Physical Sciences Research Council, and by the University of York. For this latter support, I am grateful to Prof. Peter Watson, of the Department of Electronics.

I wish to acknowledge the contribution made by the members of the Adaptive Systems Engineering Group, both past and present: Joxe Artola, Tim Brookes, Vincent Dubois, Ignacio Espert, Mike Freeman, Pete French, Neil Garner, Don Goodeve, Neil Howard, Roger Peppé, Vicky Rodriguez, Barry Scowen, Ian Stevenson, Catherine Thomas, Eugenio Zabala and Joanna Zelechowska. Working in the group has been an enjoyable and enlightening experience, and I wish you all every success in the future.

Thanks are due to my supervisor, Dr. Andy Tyrrell, whose advice has been invaluable in the completion of this work. In addition, I wish to thank my colleagues at University College London, for their patience during the completion of this thesis.

On a more personnel note, I wish to thank the following people, without whose support, advice and friendship, these last four years would have been a much less pleasant experience: Claire Faulkner, for starting it all; Jill Paterson, for being there when things went wrong; Stewart Cambridge, for meaningless coincidences; Carrie Vaughn, for patriotism; Peter Phillips, Chris Paulson-Ellis, Zoë Wood and Satty Mann for the loan of a basement; and, finally, to Sonja Krugmann, with love.

Declaration

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at the University of York. I confirm that this thesis is entirely my own work, and that all contributions from other agencies through publication or direct contact are explicitly attributed.

Extracts from this work have been presented at the 2nd Conference on the Mathematics of Dependable Systems, University of York, 4-6 September 1995 [74]; and at the 7th European Simulation Symposium, Friedrich-Alexander-Universität Erlangen-Nürnburg, 26-28 October 1995 [73].

Chapter 1

Introduction

In recent years, the use of embedded computers to provide control functionality as a part of a complete system has become commonplace. In many cases, these embedded computers are fundamental to the operation of a system, and it is not possible to control that system should the computers fail. Examples of such systems may be found in all areas: from simple embedded micro-controllers operating washing machines and toasters; to aircraft flight control systems, medical instrumentation and chemical plant control systems. Indeed, there is a growing realisation that computers can contribute substantially to accidents when they operate as a subsystem within a potentially dangerous system, and the growing reliance of society on such computer-based control is becoming a matter for serious concern.

There are two methods by which that concern may be allayed: fault prevention and fault-tolerance. If extreme care is taken in the design and implementation of a system it is possible that that system may be fault-free, and will always operate correctly: This is the approach of *fault prevention*. This is, of course, a difficult and expensive process; and the validation of systems to such high levels of dependability is the subject of some controversy [55, 56]. Indeed, many systems are of sufficient complexity that fault prevention can never be completely successful, and it must be assumed that some faults are present in the completed system. This process is highlighted by the many reliability growth models which have been developed [43, 53–55, 64, 87]; those wishing to produce systems which contain no faults are clearly fighting against the law of diminishing returns.

Alternatively, if it is accepted that a system will contain faults, effort may be expended to predict the likelihood of the occurrence of those faults, and to limit the effects of those faults which do cause errors. This is an engineering solution to the problem of designing reliable systems: produce a system which is fault free, to the extent that is *reasonably* *practicable* [28]. Once this is complete, an assessment must be made of the *tolerability* of risk from that system (for example, see [36]); and for this purpose some means of modelling the system's reliability must be employed. Following these considerations, a novel system reliability model is proposed, which provides greater expressive power in terms of derivation of the reliability and timing properties of a system, when compared with previously proposed models.

A growing awareness of the increasing reliance of society on such unproven technology, together with a number of widely publicised failures (for numerous examples, see the discussion in [1]), has lead to much effort being expended in both areas. It is, however, the second method which forms the basis of this thesis: the modelling and reliability analysis of real-time embedded systems is considered, together with techniques for achieving reliability in the presence of faults.

Following this introduction, the main body of this thesis is composed of the following chapters:

- *Chapter 2 Fault Tolerant Systems*: The problem of achieving reliability in embedded real-time systems and techniques for providing fault tolerance are discussed, together with their relative advantages and disadvantages.

- *Chapter 3 Reliability Modelling*: The need for reliability modelling to determine the effectiveness of fault tolerant systems is discussed. This is followed by a critical overview of the applicable techniques, leading to a discussion of the limitations of these techniques when applied to real-time systems, and the need for a new system reliability model.

- *Chapter 4 A New System Reliability Model*: A new reliability model is developed. This model is based on the notion of random fault occurrence and provides a stochastic view of the execution of a system, allowing the state of the system to be derived for a particular point in its operation [74].

- *Chapter 5 Generic Real Time System Model*: The model developed in chapter 4, is used to derive a generic model for the behaviour of real time systems. This model uses a generic high-level formalism based upon a Markov chain with a lattice structure which represents the progress of a computation, allowing *both* functional and time correctness of the system to be modelled. This is an improvement on traditional system reliability models, such as those discussed in chapter 3, which typically focus on functional correctness, and do not adequately model the temporal properties of such systems.

- *Chapter 6 Application to Systems Modelling*: A number of the techniques for software fault tolerance which were discussed in chapter 2 are revisited, and their reliability characteristics are evaluated using the model derived in chapter 5 [73]. This leads to the possibility of generating more accurate failure time information than has been available previously, and hence for improved reliability modelling.

- *Chapter 7 Application to Scheduling*: A discussion of the application of such accurate failure time information to the problem of scheduling real-time systems is made.

Finally, chapter 8 concludes the thesis and provides suggestions for further work. A number of appendices follow, providing further data to reinforce a number of the conclusions drawn in the main text.

Chapter 2

Fault-Tolerant Embedded Systems

It has been noted that, unless extreme care is taken in the design, implementation and validation of a system, then that system will contain faults. In addition, that system will be fault intolerant unless specially designed to tolerate faults. In this chapter, the techniques required to achieve fault tolerance are discussed, together with their applicability to different classes of system.

There are many classes of system which must be considered when designing techniques for achieving fault tolerance: the requirements of a real-time embedded control system are clearly different to those of an electronic mail delivery system, for example; yet both systems must be designed to tolerate a range of possible faults. Since it is desired to compare the properties of different techniques, the range of applicability of those techniques must first be defined: it is pointless to compare techniques designed for entirely different application areas. For this reason the discussion in this thesis will be restricted to real-time embedded systems.

There are two classes of real-time system: soft real-time systems which may tolerate occasional missed deadlines, provided that on average, deadlines are met; and hard real-time systems in which the response must occur within a specific time period or else system failure will occur: any missed deadline is fatal. As an example of this, consider a long running database system, where there is some potential for transient hardware failure. A system such as this may be designed using transaction based techniques, with a potentially unlimited retry to ensure an eventual consistent state. Whilst faults in a system such as this will degrade performance, there is no specific penalty for late results. In contrast, a flight control system must produce results which are not only correct, but also timely. In practice, there is a continuum between these two extremes of behaviour, but the distinction is useful to retain when considering the behaviour of differing techniques for achieving fault tolerance.

The remainder of this chapter will discuss differing mechanisms for building fault tolerant real-time systems.

2.1 Implementation of fault-tolerance

The basis for implementing fault tolerance is redundancy. A system is designed with excess capacity, and that excess is used to provide for the operation of the system in the presence of faults. This principle was aptly detailed by Lardner who, in 1834, described a means by which the results generated by Babbage's mechanical calculating engines could be checked:

> "The most certain and effectual check upon errors which arise in the process of calculation, is to cause the same computation to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods." [62]

The same technique is employed today, with the redundancy being implemented using some combination of hardware and software techniques. It is possible to create a system which only uses hardware fault-tolerant techniques, and of course a system which only employs software fault-tolerance is also feasible. Most real systems, however, employ some combination of the two. Each method has its advantages and disadvantages, and each will be discussed in turn.

2.1.1 Hardware based fault-tolerance

Fault-tolerant hardware has been employed in control systems for many years, and the techniques can be considered to be relatively mature. The basic concept is that of replication: if there are multiple systems performing the same task, the probability of them all failing at once should be less than the probability of a single system failing.

Most hardware fault-tolerance schemes employ a form of static redundancy. Multiple functional units are used, each receiving the same inputs and hopefully producing the same output. The outputs are compared by a voter and any incorrect results are masked out. In this context, the definition of "the same output", and of an incorrect result, must be made with care: in order to avoid common-mode failures, it is not atypical for

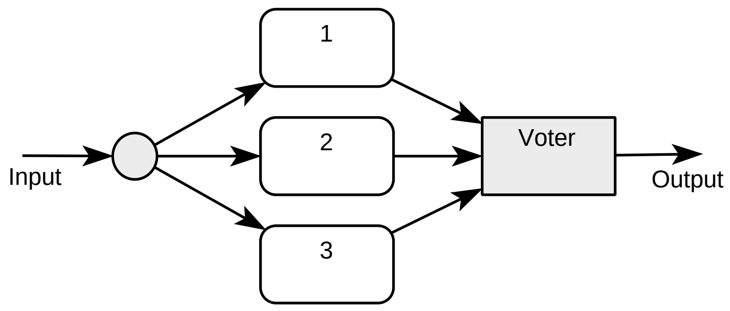

Figure 2.1: Triple-Modular Redundancy with Voter

the multiple functional units to be dissimilar, and to operate using differing algorithms, resulting in outputs which must all be considered correct, but which may not be identical. In cases such as these, some form of inexact voting must be employed.

The key to obtaining reliability in such systems is to ensure that the reduction in reliability caused by the additional hardware added, is less than the increase in reliability gained due to the ability to mask faults occurring in a minority of versions.

A common version of this scheme, known as triple-modular redundancy, is illustrated in figure 2.1. From this it should be obvious that this technique has a number of problems:

- The system has two failure points in the voter and distributor which cannot be replicated.
- Vote comparison problems can occur if different systems produce results which are similar but not identical. Consider, for example, rounding errors in a floating point calculation.
- There may be problems involving the granularity of comparisons. The potential for divergence in the results increases with the size of the replicated block, especially if the replicated blocks have differing designs in order to avoid common-mode failure.

Despite these problems this technique, or variations on it, is widely used in many commercial systems [11, 12, 19, 20, 24, 41, 84, 89, 92].

This then is the basis of static hardware fault tolerance. Systems are replicated a number of times with *voters* used to detect failure and reconfigure the system so as to maintain correct operation. This technique has been modified and enhanced in many ways and under many names, but the basic concept remains as described.

This thesis will not consider hardware fault tolerance in any greater detail since it is felt that these techniques are sufficiently well known and well developed that further study would not be appropriate.

2.1.2 Software based fault-tolerance

When compared with the hardware based techniques mentioned above, software based fault-tolerance is a relatively underdeveloped field of research. Hardware fault-tolerance requires expensive custom hardware to achieve its fault-tolerant properties, the aim of software fault-tolerance is to use standard hardware and to move the fault-tolerant properties of the system to software control, the greater software complexity is offset by simpler and cheaper hardware systems.

The next section of this thesis will discuss some of the techniques used to achieve software fault-tolerance in more detail. The remainder of this section will be devoted to an overview of the concepts involved in designing such systems.

The first point to note about software fault-tolerance is that it is not necessary to have *any* fault tolerance built into the system hardware to employ software fault-tolerant techniques. In many cases system failures are caused not by the hardware breaking down, but by faults in the software, and in such cases software fault tolerance can prevent many problems. There are many techniques which can be used to provide recovery in the event of software failure, and it is important to distinguish the applicability of each to the many differing failure modes of a system. As an example of this, it is useful to discuss the differences between forward and backward error recovery techniques.

Forward error recovery schemes must manipulate the current system state in order to perform error recovery [3]. This is typically implemented using exception handling techniques [13] and other application specific procedures. A forward error recovery scheme has the advantage of working from the current system state, and so it can be made very efficient in many cases. However, it also has the problem that any recovery mechanism must necessarily be very application specific, and so forward error recovery cannot be used to recover from unanticipated errors.

In contrast, backward error recovery schemes [3] rely on restoring a prior system state which will, hopefully, be free from error. Backward error recovery is typically implemented as some form of recovery block or atomic transaction scheme (See, for example, [26, 44] for a discussion of recovery blocks, and [91] for a discussion of transaction based techniques). The advantage of backward error recovery is that it is a generalised approach, and can recover from any software fault, anticipated or not, provided that the software does not access unrecoverable external objects[1]. The disadvantage is that a large overhead may be incurred when saving the system state.

These two mechanisms for implementing software fault-tolerance indicate the importance of selecting the correct recovery scheme for the application. If the system designer is aware of all possible system failure modes and efficiency is important then a forward error recovery scheme is appropriate, but in other cases this may not be so. The designer's task is further complicated if multiple software failures can occur simultaneously or if it is necessary to include the possibility of hardware failure in the system.

Whilst software faults can be countered using software based fault-tolerance, it should be obvious that a system which relies solely on fault-tolerant software is forever at the mercy of hardware failure. Fortunately the provision of software fault-tolerance can greatly reduce the complexity of the hardware required to achieve full fault tolerance. Instead of employing special purpose hardware it is possible to use general purpose *off the shelf* hardware and use software to perform the functions of distribution and voting on the results. It is therefore seen that an unreliable, but replicated, hardware system (comprising off-the-shelf components) can be made reliable by using a layered software protocol [2], that is, a protocol to make a reliable system at level n from an unreliable system at level $(n-1)$. Hardware failures are masked using software techniques, resulting in a reliable system without the special purpose hardware. There are several methods for achieving such reliability, examples include *atomic transactions* (section 2.3.1), *N-self checking programs* (section 2.2.2) and *distributed conversation* (section 2.3.5) schemes. These are discussed in more detail in the remainder of this chapter.

2.2 Basic Techniques For Software Fault-Tolerance

The basic types of fault-tolerance available at the application level are N-version programming, N-self checking programming and atomic actions. These three methods are widely used, and can be extended to provide virtually all software fault-tolerant schemes in use today. Each will be discussed in turn.

[1] An unrecoverable external object, otherwise known as a non-program object, is some object or device controlled by a software system, yet not part of that system. Changes to such an object are generally irreversible, hence recovery once such an object has been modified is generally not possible, or extremely difficult. Examples include: aborting the firing of a missile, reversing a chemical reaction in a control plant, etc. There has been some work conducted which uses postponed execution schemes to reduce the effects of non-recoverable objects, and to allow limited rollback recovery in such cases [26, 80]; the problem is, however, a fundamental one for systems interacting with external devices, and there are no real solutions.

2.2.1 N-Version Programming

N-Version programming is the software equivalent of N-Modular redundancy in hardware design [13, 49]. It requires the generation of N versions of a program block, each created independently of the others. The blocks are executed concurrently, with the same inputs, and a voter process compares results and passes the consensus result to the rest of the system. N-Version programming is based on the assumption that the program can be specified in such a way that each version performs the same task, and that independent designs will not have common mode failures; this is known as *design diversity*. To further avoid the possibility of common failure in one, or more, versions it is common for the different versions to be run on different processors of a multi-processor system. Such a scheme will also, of course, reduce the probability of a hardware failure causing a complete system failure.

N-version programming has the advantage that it provides fast result switching (because of the parallel execution of the different versions), but it also suffers from the same problems as the N-modular redundancy techniques used in hardware fault-tolerance (see section 2.1.1). In addition, it is often difficult to design multiple software systems which have the same specification, but use differing algorithms. As a consequence of this, common-mode failures can be frequent in N-version programming systems [8, 47, 52].

2.2.2 N-Self Checking Programming

N-Self Checking Programming, NSCP, is defined by Laprie [49] as follows:

> "...a self-checking program results from adding redundancy to a program so that it can check its own dynamic behaviour during execution. A self-checking software component consists of either a variant and an acceptance test or two variants and a comparison algorithm."

The NSCP scheme consists of multiple variants of a program block executing in parallel. During normal execution, only the primary block provides output and the alternates remain as "hot" spares. If a failure occurs in the primary, one of the alternates begins providing output. Unlike the N-version programming scheme, where all variants provide output and a voter determines the result, the NSCP scheme relies on the detection of the error in the primary and subsequent result switching. Each variant in the NSCP scheme has its own acceptance test, which determines if its result is acceptable, in the N-version programming scheme the decision is cooperative.

The advantages of NSCP are that it provides relatively fast result switching when an error occurs, and it is possible to distribute the variants among multiple processors in order to provide resilience to hardware failure. A possible disadvantage of NSCP is that each alternate decides the acceptability of its own result: there is no independent checking of a module's output.

2.2.3 Atomic Actions

An atomic action is an operation which, as far as other processes in the system are concerned, appears indivisible and instantaneous. The processes performing the action can detect no state change except that performed by themselves, and they do not reveal their state changes until the action is complete. There are a number of requirements for an action to be classed as atomic [13]:

- The action must have well defined boundaries.
- An atomic action must not allow the exchange of information between processes active in the action and those outside.
- Only strict nesting of atomic actions should be allowed.
- It should be possible to execute different atomic actions concurrently.
- Atomic actions should allow recovery to be programmed.

It should be noted that atomic actions do not, in themselves, implement fault-tolerance. They simply restrict communication patterns, allowing other constructs to build recovery procedures on top of them. Atomic actions are the basis for a number of useful techniques, such as atomic transactions, recovery blocks and conversations. These are discussed in more detail in section 2.3.

For a further discussion of atomic actions the reader is referred to [3].

2.3 Advanced Techniques for Software Fault Tolerance

The discussion in section 2.2 has focused on the basic building blocks for software fault tolerance. In many cases, these simple techniques are not sufficient, and must be extended to provide the required properties. A number of common extensions to these basic techniques will now be discussed, and their advantages and disadvantages highlighted.

2.3.1 Atomic Transactions

An atomic transaction [31, 63] is an extension to the atomic action (section 2.2.3) which supports failure atomicity. An atomic transaction will either complete successfully or it will have no effect; it cannot partially complete and leave the system in an inconsistent state. As has been noted, the atomic action by itself does not provide fault-tolerance, rather it restricts communication patterns so that recovery procedures can be built above it. The atomic transaction is the first such recovery procedure which can be built, and as such it provides an important stepping stone between the low-level atomic action and higher-level techniques. This is an example of the layering protocols described in section 2.1.2.

The atomic transaction technique can be implemented for program objects with a minimum of run-time support. This is typically done by application level calls to a `BeginTransaction` primitive which saves the system state, and an `EndTransaction` primitive which discards the saved state. If a failure occurs between these calls the system state can be restored, and the transaction can either be aborted or retried easily. The `BeginTransaction` and `EndTransaction` primitives can either be inserted by the application level programmer, or can be generated automatically by the compiler.

For non-program objects of an information nature, for example files or databases, the transaction scheme can be implemented in a similar manner, so long as all processes involved can be determined, and the objects themselves are able to save previous values of their state to enable rollback on failure. Many such systems have been implemented, for example, [85], most involving database management or network file systems.

The final class of object for which transactions could be required is the non-program object of a physical nature; that is, an external hardware device. As discussed previously (footnote on page 8), the recovery of such objects is often difficult, since many actions performed by these objects are irreversible. The atomic transaction may sometimes be employed in these cases, when combined with a postponed execution scheme: failure atomicity is achieved by delaying certain actions until the remainder of the transaction is guaranteed to succeed. This is discussed further in [39, 67, 80, 91].

2.3.2 Recovery Blocks

The recovery block [78] is a technique which uses multiple versions of a program block to attempt to ensure success in the presence of system failures. The syntax of the recovery block is typically given as follows:

```
ensure <acceptance test>
by
  <primary module>
else by
  <alternative module>
else by
  ...
else
  error
end
```

Each alternate module is constructed from an atomic transaction which provides a complete and simple capability to undo the effects of any alternate module. The program proceeds by executing the primary module followed by the acceptance test. If the acceptance test succeeds then the recovery block is exited. If the test fails, the effects of the primary module are undone, and the first alternate module is executed. This repeats until one of the alternates passes the acceptance test, or until the final alternate fails, at which point the entire recovery block fails and its effects are undone.

It can therefore be seen that the recovery block as a whole performs as an atomic transaction, with the advantage of having design diversity as part of its structure.

As described in [13] there are a number of features of recovery blocks which should be noted, these include

- *Overheads*: In many cases it has been found that checkpointing can be expensive in both time and storage required. This may be a problem in certain time-critical applications, although it is not felt that the problem is necessarily more severe than with other recovery mechanisms. When compared with N-Self checking programming (NSCP), for example, the overhead due to checkpointing is countered because NSCP requires multiple alternates to execute concurrently.

- *Atomicity*: As with the atomic transaction, it is difficult to include processes which interact with the non-program environment in the recovery block due to the difficulties involved in restoring the state of non-program objects of a physical nature.

- *Dynamic redundancy*: One problem with the recovery block structure is that the alternates are executed sequentially, and because of this there can be considerable delay and non-determinism incurred during error processing. This property is inherited from atomic actions and atomic transactions. When compared with N-Self checking programming and N-Version programming this can seem a considerable disadvantage for real-time systems, and for this reason the distributed recovery block, section 2.3.3, has been developed.

The recovery block concept is firmly rooted in uniprocessor systems and, as such, it cannot easily be used to recover from processor (or other hardware) failure, since the possibility of executing the alternates on multiple processors is not present. For this reason, and because of the problem of slow recovery mentioned above, the basic recovery block scheme is being superseded by the distributed recovery block. For systems where the probability of hardware failure is low, however, the recovery block is still an attractive concept, to catch software faults.

2.3.3 Distributed recovery block

The distributed recovery block is an extension to the recovery block scheme in which the alternate modules execute in parallel. The acceptance test is replicated on multiple processor nodes, and the alternates are similarly distributed. All alternate modules execute concurrently and evaluate their acceptance tests. If the primary node passes its acceptance test it passes the result out and informs the alternates, which do not perform output. If the primary fails its acceptance test it informs the first alternate, which passes its result out, provided that it has passed its acceptance test. This process is further described in [44, 45].

The distributed recovery block can be seen to provide a faster response to errors than the standard recovery block scheme, although this is bought at the expense of reduced performance in the absence of errors (since it requires multiple alternates to execute in parallel). It is noted that each component in the system makes the decision on the acceptability of its result independently and informs the other alternates: the decision making is not cooperative. In addition, the distributed recovery block scheme will cope with processor failure provided that some sort of time-out mechanism is employed.

2.3.4 Conversation

The conversation is a recovery scheme based upon both the recovery block and the atomic action. The recovery block enables recovery to be programmed for a single process, and the atomic action is a means of controlling inter-process communication. When combined they provide a means of programming recovery for a group of interacting processes. The conversation is seen to be an atomic action involving multiple processes with one, or more, of the processes involved in the action having recovery blocks built in to their structure.

The semantics of the conversation can be summarised as follows:

- On entry to the conversation all processes must save their state. This provides for recovery should the entire conversation fail.
- Whilst in the conversation, inter-process communication is restricted in the same way as for an atomic action.
- In order for any process to leave the conversation, all processes must have passed their acceptance test.
- If any process fails its acceptance test all processes involved in the conversation have their state restored and execute their alternate module.
- If all alternates fail the entire conversation is abandoned and recovery must be performed at a higher level. The conversation as a whole exhibits failure atomicity.
- Only strict nesting of conversations is allowed.

The syntax for a single process taking part in a conversation is typically presented as follows [13]:

```
action A with (P2, P3) do
  ensure <acceptance test>
  by
    <primary module>
  else by
    <alternative module>
  else by
    <alternative module>
  else by
    ...
  else
    error
  end
end A;
```

The other processes involved in the conversation are declared similarly. It can be seen that this is similar in structure to the syntax of the recovery block.

It is noted that since conversations are implemented using atomic transactions it must be possible to completely undo the effects of any operation performed during the conversation. As has previously been noted this is problematic in cases where the program must interact with the non-program environment. In particular, the following points should be noted

- Since all effects of the conversation must be undone in the event of a failure, it is not possible for a process active in a conversation to affect the non-program environment in an unrecoverable manner unless it is known that all processes in the conversation have successfully passed their acceptance tests.
- It is difficult to build nested conversations easily, since the inner conversation must be restored if an outer conversation fails. Hence any interactions with the non-program environment performed by a nested conversation must be recoverable, or else they must be delayed until all outer conversations have completed.

It is seen that a conversation should only be permitted to change the non-program environment in an unrecoverable way when it has successfully completed, and because of this nested transactions which affect the non-program environment must be programmed with care. For a further discussion of this the reader is referred to section 4.1 of [91].

A further problem is inherent in the multi-process nature of the conversation technique: the *domino effect*. This problem arises when multiple processes, which may be subject to rollback recovery, exchange information. Consider two processes, A and B, each comprising a recovery block structure; and further assume that process A communicates some information to B during its execution. If process A fails its acceptance test, and has to rollback its execution, and retry; it is clear that process B must do so too, since process B has based its results on faulty data passed from A. In the general case there may be multiple processes interacting, and this effect may ripple back through the entire process set. It is possible to avoid this problem, by ensuring consistent recovery points are set up, to ensure process synchronisation. This leads to additional overhead, however.

The conversation scheme, or some similar technique, is of great importance in the design of fault tolerance for parallel and distributed systems. The problems noted are not insurmountable, and the advantages of concurrent execution typically outweigh the disadvantages caused by the extra synchronisation work needed to prevent the domino effect.

2.3.5 Distributed conversation

In a similar manner to the distributed recovery block it is possible to extend the conversation scheme to allow for different processes to reside on multiple processors. This is known as the *distributed conversation scheme* and can be considered to be a unification of the concepts of the conversation and the distributed recovery block.The distributed conversation has an advantage over the normal conversation because it is able to recover from processor (or other hardware) failure in a transparent manner, provided the acceptance tests include a timeout provision.

The distributed conversation is discussed further in [44, 45, 68].

2.4 Summary

This chapter has discussed the basics of fault-tolerance for real-time embedded systems. The necessity of introducing redundancy into the implementation of a system, as the basis for fault-tolerance, was discussed in section 2.1; and it was shown that both redundant hardware and software may be employed in the production of fault tolerant systems, and that, typically, some combination of techniques is employed.

Hardware based schemes were discussed in some further detail in section 2.1.1, where the wide application of these techniques was noted. Since such techniques are well understood, it was decided that further study in this area would not be appropriate.

Section 2.1.2 discusses the basics of software based fault tolerance. This is extended by sections 2.2 and 2.3 where specific design techniques for implementation of software-based fault tolerance are discussed.

These techniques will, if applied correctly, allow for systems to be constructed which can tolerate the effects of a wide class of faults and operate within strict performance criteria. Before this can be achieved, however, it is necessary to define those performance criteria, and to employ an accurate predictive model of the system's behaviour to allow effective design. This is discussed further in chapter 3.

Chapter 3

Reliability Modelling

Before fault-tolerant features can sensibly be applied to a system, there is a need to determine the effects they have on the reliability and failure modes of the system as a whole. In particular, it is important that an accurate failure/reliability model is available during the design of fault-tolerant and safety critical systems, whether those systems comprise hardware, software or some combination of the two.

The reliability models which have been developed in the literature may be split into two broad groups: functional models which describe the system from a time-independent viewpoint, and dynamic models which describe the run-time behaviour of a system. Time-independent, functional, models [6, 38, 77, 81] are typically based around a Markov-chain or other stochastic process which is used to describe the behaviour of the system, either neglecting information about execution time or providing a partial ordering of events only. Such models enable the probability of failure for a particular structure to be calculated, but do not provide for the calculation of the timing properties of the system. Whilst this is undoubtably of value, its usefulness in the analysis of real-time systems must be questioned, since these systems require not only functionally correct behaviour, but also *temporally* correct behaviour. The timing properties of a real-time system are as important as its functional properties in ensuring correct operation and, unfortunately, this class of model is not able to describe this with sufficient rigour.

In contrast, time-dependent models are much less well developed [23, 33]. Although some work has been conducted into finding algorithms to derive the mean execution time of a set of processes in the presence of failure [79], there has been little work undertaken to determine the probability distribution of the system's execution time. Much of the research conducted with real-time systems has focused on scheduling problems [9, 35, 72, 82, 93], and typically requires knowledge of the execution time bounds of a process to enable efficient schedules to be calculated. With the introduction of fault-tolerant procedures, the execution time bounds of the system will change. It is therefore important that a means of deriving an expression for the execution time of a system with fault-tolerant processes is found.

The remainder of this chapter discusses a number of techniques by which system reliability models may be derived. In particular, techniques based around Markovian models, and Petri nets are discussed, since it is these models which have been most widely applied in the literature. The relative strengths and weaknesses of these techniques are discussed, and it will be shown that, whilst these techniques have wide applicability, they must be further developed in order to model certain classes of system.

3.1 Markov chain models

A Markov chain consists of a set of places with probabilistic transitions between them. The state of the system is represented by the probability distribution of a random variable among those places at a time t, which may be either discrete or continuous valued. Given the initial probability distribution of the random variable among the system places, and the transition probabilities between the places, it is possible to determine the probability distribution of the system at any future time. A brief formal definition of the theory of Markov chains is provided below; for a more thorough description the reader is referred to the works of Chung [18] and Takács [88]; for example.

3.1.1 Formal definition of a Markov chain

The definition of a Markov chain builds from probability theory via a set of mutually exclusive and exhaustive events, $E_1, E_2, \cdots, E_N$. These events are known as the *states* of the Markov chain.

A number of trials are performed, each of which will result in the occurrence of one of the events E_j. The set of random variables x_n is defined such that $x_n = j$ if E_j is the outcome of the n-th trial.

If the outcome of each new trial depends on the outcome of the directly preceeding trial, but is independent of the outcomes of all former trials, the set of trials form a Markov chain. This is denoted by equation 3.1, representing the *Markov property*.

$$P\{x_n = j \mid x_0 = i_0,\ x_1 = i_1,\ \ldots,\ x_{n-1} = i_{n-1}\} = P\{x_n = j \mid x_{n-1} = i_{n-1}\} \tag{3.1}$$

Based on the outcome of the trials, a system represented by a Markov chain may be viewed as a state machine; with probabilistic transition between the states. In particular if $x_{n-1} = i$ and $x_n = j$ we say that the system made a transition from state E_i to state E_j at the n-th (time) step. The conditional probabilities $P\{x_n = j \mid x_{n-1} = i\}$ are known as the *transition probabilities* of the Markov chain.

A Markov chain is *homogeneous* if the transition probabilities are independent of n. In this case it is usual to denote the transition probability, $P\{x_n = j \mid x_{n-1} = i\}$, by the more compact notation, $p_{i,j}$. Since all commonly applied Markov models are homogeneous, discussion of inhomogeneous Markov chains will be omitted.

A simple extension to the transition probabilities is to determine the probability that a transition taking n-steps occurs. An expression for this is given in equation 3.2; it may be seen that this definition is recursively derived from a number of single step transitions.

$$p_{i,k}^{(n)} = P\{x_{m+n} = k | x_m = i\} = \sum_{j=0}^{N} p_{i,j} p_{j,k}^{(n-1)} \tag{3.2}$$

where

$$p_{j,k}^{(0)} = \begin{cases} 0 \text{ if } j \neq k \\ 1 \text{ if } j = k \end{cases} \tag{3.3}$$

and

$$p_{j,k}^{(1)} = p_{j,k} \tag{3.4}$$

It may be seen that, in a homogeneous Markov chain, these n-step transition probabilities do not depend upon m.

The probability distribution $P\{x_0 = j\} = P_j(0)$ of the random variable x_0 is called the *initial distribution*. Given the initial distribution and the transition probabilities it is possible to determine the probability distribution for each random variable, x_n; this is the probability that the system is in the state E_j at time n; and is given by equation 3.5.

$$P_j(n) = \sum_{j=0}^{N} P_i(0) p_{i,j}^{(n)} \tag{3.5}$$

This probability distribution represents the state of the system.

3.1.2 Application of Markov models

Many systems can be modelled using state-based techniques, and a Markov chain is a suitable method of analysis for a subset of such systems. There are a number of reasons why a pure Markov model is insufficient to model all such state-based systems:

- As specified by equation 3.1, a Markov chain is a *memory-less* system. The past state of the model does not influence its future behaviour, except that the current state dictates the next state. This restricts the class of system to which Markov models may be applied.
- There is no explicit concept of concurrency in a Markov chain. Although this may be simulated by partitioning the chain into separate segments, the approach is clumsy; and difficult to work with. The modelling of synchronisation issues is difficult.

Models based around Markov chains are limited by this lack of descriptive power. Whilst it is possible to implicitly model the features mentioned above, the resulting systems are typically difficult to analyse: both from a computational point of view, and because of the large amount of information which is *not* explicitly stated in the model.

This does not, of course, prevent the application of the techniques of Markov chain theory being applied to a wide class of problem: for many models these techniques are ideal. Furthermore, it has been shown that Markov chain models form the underlying basis for many more advanced classes of model. That is, many such models may be transformed into an equivalent Markov chain system. The theory of Markov chains must, therefore, be viewed as a basis upon which further work is built. Many systems exist which may be modelled using the theory of Markov chains. Although such Markov models may be extremely inefficient in many cases, leading to the application of higher order modelling techniques.

The remainder of this section discusses a number of Markovian reliability models, in order to provide some measure of the scope of applicability of these techniques.

3.1.2.1 Arlat et al. and Pucci

The work of Arlat *et al.* [5] provides a good example of a Markovian model for a recovery block system. A set of failure events is derived, and a Markov chain built to consider the execution of the system, in terms of those events, with states of the chain representing each stage of the system's execution and operational mode. The number of states in the Markov chain is small, and the model derived is high-level, considering only the possible failure modes, and not their timing properties. The work of Arlat *et al.* is discussed further in section 6.4.5.

The model of Pucci [77] is similar in scope, although the precise failure events chosen and the structure of the derived Markov chain differ. It is discussed further in section 6.4.2.

Together, these models illustrate an important class of Markovian system reliability model. The high-level properties of a fault-tolerant system are modelled as a (small) set of states in a Markov chain. Such models are able to provide an estimation of the reliability of a system, provided system failures can be classified in a manner consistent with the model. Timing properties of the system are typically not modelled, except in the case of the mean number of executions of the system before failure.

A similar class of model may be used to predict the behaviour of concurrent N-version programming systems, as is also discussed by Arlat *et al.* [5]. The lack of explicit support for concurrency in Markov chain models is not a limiting factor *for this class of model*, since the concurrency of the underlying system is not represented directly in the model, but subsumed into the set of high-level failure events.

3.1.2.2 Ledoux & Rubino

The work of Ledoux & Rubino [51] discusses the limitations of Markov models as applied to the *structural* modelling of software systems. A software system is typically composed of a number of modules, with control passing between the modules according to a number of variables, including the data-set applied to the system, and the occurrence of faults within that system. It is therefore possible to derive a stochastic process representing the transfer of control between the different modules of the system, and this process is typically assumed Markovian.

Unfortunately, the memory-less Markovian assumption is found to be unrealistic in many cases; for example [51]:

> "...suppose that module M_i receives the execution control sometimes from module M_j, sometimes from module M_k. In many situations, the conditions under which the control is transferred and the characteristics of the tasks that must be performed by M_i may be completely different in the two cases. If M_i is a state of a Markov process, then the random variable "nth sojourn time in M_i" is independent of the other sojourn times (in M_i and in the other states), does not depend on n and does not depend on the identity of the module from which the control is got. Hence, such a Markovian assumption may be too strong to be acceptable."

Ledoux & Rubino circumvent this problem by providing additional states in their model, such that a single state is replaced by several states, one for each possible path by which that single state may have been reached. The disadvantage with an approach such as this is the large increase in the size of the state-space caused.

3.1.2.3 Csenki and Balakrishnan & Ragavendra

The work of Csenki [21] provides a further example of the problem of state-space explosion inherent in the use of Markov models. Once again, a model for the reliability of a recovery block system is derived. In this case, however, the author attempts to model the influence of nested clusters of failure points; resulting in the derivation of a Markov model with a unbounded and discrete state space. In this case, the structure of the model allows for approximate solutions to be derived; other Markov models, such as that discussed by Balakrishnan & Ragavendra [10] have similar problems of state-space explosion, and various special techniques are employed to derive computationally feasible solutions.

It is clear, from models such as these, that basic Markov techniques are limited by the problems of state space explosion, and that unless care is taken in the definition of the model, solution may become computationally infeasible for all apart from small systems.

3.2 Petri Net Models

A Petri net model [75, 76] consists of a set of places, together with a set of transitions. These are interconnected by a series of directed arcs, which enable tokens to move from place-to-place by means of the transitions. Movement of the tokens is defined by the firing rules of the network. In a simple Petri net it has been noted that:

> "The use of the tokens rather resembles a board game. These are the rules: Tokens are moved by the *firing* of the transitions in the net. A transition must be *enabled* in order to fire. (A transition is enabled when all of its input places have a token in them). The transition fires by removing the enabling tokens from their input places and generating new tokens which are deposited in the output places of the transition." [75]

The placement of tokens in the network defines a *marking* which represents the state of the system modelled.

A Petri net model, Q, may be defined formally as a set of places, P, a set of transitions, T, input and output arcs, A, and an initial marking, M_0 [61]:

$$Q = (P, T, A, M_0) \tag{3.6}$$

where

$$P = \{p_1, p_2, \cdots, p_n\} \tag{3.7}$$

$$T = \{t_1, t_2, \cdots, t_m\} \tag{3.8}$$

$$A \subset \{P \times T\} \cup \{T \times P\} \tag{3.9}$$

$$M_0 = \{\mu_1^0, \mu_2^0, \cdots, \mu_n^0\} \tag{3.10}$$

At a particular instant, the marking may be viewed as a mapping from the set of places, P, to the natural numbers, $\mathbb{N}$:

$$M : P \to \mathbb{N} \text{ where } M(p_i) = \mu_i \text{ for } i = 1, 2, \cdots, n \tag{3.11}$$

A marking, M_1, is reachable from a marking, M_2, if there exists a sequence of transitions, t^*, such that $M_2 \xrightarrow{t^*} M_1$. The reachability set, S, is the set of all markings, $\{M_i\}$, which are reachable from some initial marking, M_0. This reachability set defines the possible states which the system modelled may enter; calculation of this is therefore a major activity in the Petri net modelling of systems.

Petri nets are of use in the modelling of concurrent, discrete event, systems. In particular, they allow the modelling of events and conditions, and the relationships between them. The occurrence of events is modelled by the firing of transitions. Since a transition will not fire until enabled, and cannot be enabled until all its input places have a token in them, synchronisation is easily modelled. Concurrency may be modelled, since a transition may have multiple outputs, and hence generate tokens in several places when it fires.

It can be seen that a basic marked Petri net system is able to model both the static and dynamic properties of a system. The basic Petri net model does, however, have a number of limitations. In particular, it is both asynchronous and non-deterministic. This has two important consequences:

1. Because of the asynchronous nature of Petri net systems, it is noted that:

   > "There is no inherent measure of time or the flow of time in a Petri net. This reflects a philosophy of time which states that the only important property of time, from a logical point of view, is in defining a partial ordering of events." [75]

   That is, a Petri net system will allow the *logical* properties of a system to be described, but it cannot provide a model of the temporal properties of a system, except that it can define a partial ordering of events.

2. A Petri net model of a system consists of a sequence of discrete events whose order of occurrence is defined by the movement of tokens around the system by means

of a set of *transitions*. If, at any time, there are multiple enabled transitions, then any *one* transition may fire. The choice of transition is made in a non-deterministic manner, and it is noted that the firing of a transition is considered instantaneous, and hence multiple transitions *cannot* fire simultaneously.

These features combine to make the basic Petri net model suitable for use in modelling the qualitative properties of systems, but restrict its usefulness in modelling the quantitative behaviour of systems. It is, for example, possible to determine reachability of a particular system state using a basic Petri net system; but it is not possible to determine the probability that that state is reached at a particular time. In order to fully model the quantitative properties of a system it is necessary to add some form of temporal specification to the basic, untimed, Petri net model: such extensions are discussed in sections 3.2.2 to 3.2.5.

3.2.1 Coloured Petri Nets

A coloured Petri net model results from a desire to reduce the size, and hence the complexity, of Petri net models. This is done by folding similar places and transitions into single entities and having tokens in the net represent different values, or *colours*. Transitions in the net are labelled by the colour set which will trigger their firing, leading to a system where the behaviour of the net is different, depending on the colour of the tokens present. This enables a significant optimisation of a number of classes of model, resulting in a much reduced state space. It is noted that coloured Petri nets are isomorphic to standard Petri net models: they are an optimisation only, and do not enable analysis of additional classes of system. For this reason, coloured Petri nets are not studied further here.

3.2.2 Timed Petri Nets

Timed Petri nets [59] are an extension to the basic Petri net model in which a firing delay, which is assumed to take one of a set of discrete values [61], is associated with each transition. In such a system, transitions can be viewed as firing in three phases:

1. A *start firing phase* when tokens are removed from the input places.
2. A *firing in progress phase* associated with the firing delay, where tokens are assumed to be processing in the transition.
3. An *end firing phase* when tokens are deposited in the output places.

A system such as this will allow the temporal properties of a system to be modelled, and indeed the use of such systems is noted a number of times in the literature (see, for example, [4, 15]). It is, however, noted in [57] that the timed Petri net model is effectively a sub-class of the stochastic Petri net model, and therefore, it will not be discussed further here.

3.2.3 Stochastic Petri Nets

Stochastic Petri nets [14, 34, 57, 60] are an extension to the basic Petri net model where a firing delay is associated with each transition. Firing delays are random variables with negative exponential probability density function. When a marking is entered, each enabled transition samples this probability function, and the transition with the lowest firing delay will fire to determine the new system marking.

The basic Petri net model, defined in section 3.2, is, therefore, extended to give a stochastic Petri net by the addition of the set of transition rates, $\lambda = \{\lambda_1, \lambda_2, \cdots, \lambda_n\}$ for the exponentially distributed firing times. The probability, $p_i < 1$, is defined as the probability that the enabled transition, t_i, fires at a particular time step, given that no other transition fires. These values may be determined from the average firing delay, $\bar{\tau}_i = 1/\lambda_i$.

It is noted in [57, 60] that a stochastic Petri net model with this firing delay is equivalent to a continuous time Markov chain model. This opens up a new area of analysis, allowing performance measures to be calculated, based on an analysis of the underlying Markov model. The use of stochastic Petri nets can be seen to allow the modelling of the temporal properties of certain classes of system. There are two reasons, however, why they are not a general solution to this problem:

- There are certain classes of system which can be modelled simply using stochastic Petri nets, but which have an infinite number of states when converted into their corresponding Markovian form [57]. Analysis of systems such as these is difficult, using standard Markovian techniques. Further, the underlying Markov chain is typically much larger than the Petri net system: For some systems, analysis may be computationally infeasible.
- Markovian analysis requires the "memoryless" property. That is, if a transition fires to change the marking of the net, the distribution of the time remaining on the other enabled transitions is not affected [61]. For some systems this restriction is not acceptable.

These restrictions are removed by the discrete time stochastic Petri nets, section 3.2.4, and by the so called generalised stochastic Petri net, as discussed in section 3.2.5.

3.2.4 Discrete Time Stochastic Petri Nets

The discrete time stochastic Petri net [61], bridges the gap between timed Petri nets and stochastic Petri nets. Unlike a continuous time stochastic Petri net, the discrete time stochastic Petri net only allows transitions to fire at certain times, hence opening the possibility of several transitions firing at once. As Molloy [61] notes:

> "Since multiple firings may occur at any time step, the probabilities for each possible combination need to be determined. Requiring a designer to actually assign those probabilities is too difficult. The definition of discrete time stochastic Petri nets [...] takes a different approach. Taking an approach similar to circuit analysis tools, we [...] specify the probability that a transition would fire, once enabled, at the next time step given the fact that nothing else would happen (ie: no other transition would fire). This idea of assigning the conditional probability takes the burden off the designer and places the deconditioning calculation into the analysis (usually automated)."

The downside of this is that the analysis is more complex than that required for continuous time stochastic Petri nets. Molloy further notes that this *does not* increase the size of the underlying Markov chain, but it does make the transition matrix more dense, limiting the usefulness of sparse matrix techniques as a shortcut method of analysis.

The definition of a discrete time stochastic Petri net follows from the definition of a basic Petri net, section 3.2, in a similar manner to the definition of a continuous time stochastic Petri net. A set of geometrically distributed firing times, $\rho = \{\rho_1, \rho_2, \cdots, \rho_m\}$, are added to the model. The probability, $p_i < 1$, is defined as the probability that an enabled transition fires at the next time step, given that no other transition fires. These values are obtained by determining the average firing delay, $\bar{\tau}_i = \Delta\tau / p_i$. It is noted that the limiting case of the discrete time stochastic Petri net, as $\Delta\tau \to 0$ is a continuous time stochastic Petri net.

3.2.5 Generalised Stochastic Petri Nets

The generalised stochastic Petri net, GSPN, [17, 58] is a further extension of the basic Petri net model. Like the stochastic Petri nets discussed in section 3.2.3 the GSPN includes transitions with exponentially distributed firing delays. In addition, a GSPN allows immediate transition, which fire immediately when enabled. These immediate transitions will fire in preference to timed transitions if both are enabled for a specific marking. If multiple immediate transitions are enabled, a *switching distribution* specifies the probability that each particular transition fires. It is noted that this switching distribution may result in a transition which is enabled but has zero probability of firing: this transition must behave as if it is not enabled.

A useful optimisation, in such a case, is to define *inhibitor* transitions. These function in the opposite manner to normal, preventing firing if tokens are in their input places. It is noted that such inhibitor transitions do not increase the modelling power of a GSPN: they are an optimisation only, based on adjustment of the switching distribution for certain immediate transitions.

It has been shown that GSPNs are still equivalent to Markov models. When transformed into the equivalent Markov model, standard analysis may be performed, allowing for performance evaluation of GSPN models. In addition, the structure inherent in a GSPN model maps down to the underlying Markov model, and may enable optimisation in the analysis techniques.

3.2.6 Petri Nets: Summary and Examples

The techniques discussed in sections 3.2 to 3.2.5 allow for the modelling of concurrent, discrete event simulations. In particular, they allow the modelling of events and conditions, and the relationships between them. The occurrence of events is modelled by the firing of transitions. Since a transition will not fire until enabled, and cannot be enabled until all its input places have a token in them, synchronisation is easily modelled. Concurrency may be modelled, since a transition may have multiple outputs, and hence generate tokens in several places when it fires.

The various extensions to the basic Petri net allow for modelling the timing properties of a system, with varying degrees of descriptive power. The essential basis for these extensions is an underlying Markov model, hence it is clear that the different techniques are essentially equivalent in the range of systems which may be modelled. Where the different extensions to the basic Petri net model differ is in the ease with which certain behaviour may be modelled.

The remainder of this section will present a number of examples, to illustrate the means by which these techniques have typically been employed for reliability modelling. This can clearly not be an exhaustive survey: the range of Petri net models studied in the literature is huge. Rather, a sample of the Petri net techniques applied to reliability modelling is presented.

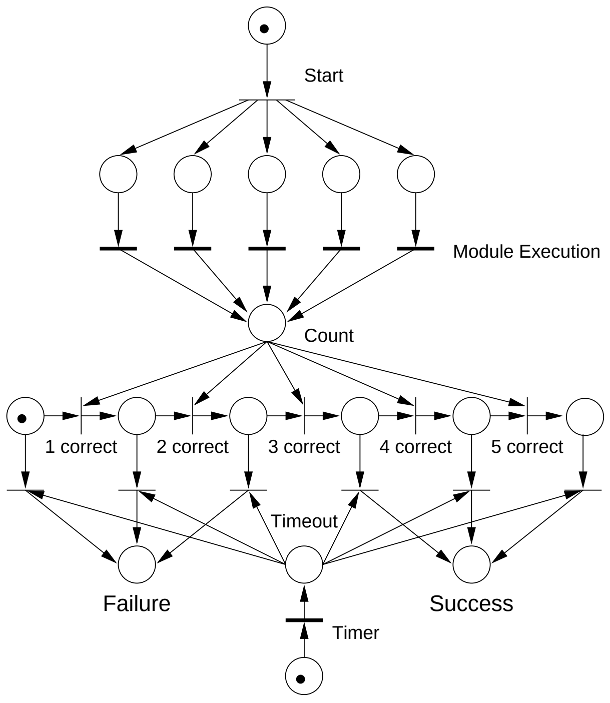

Figure 3.1: The model of Geist *et al.*

3.2.6.1 Geist et al.

The work of Geist *et al.* [30] uses a stochastic Petri net to model the synchronisation structure of N-version software. This model is illustrated in figure 3.1. In this figure, the thin bars represent instantaneous transitions, whilst the fat bars represent timed transitions. The first portion of the model, between *start* and *count* represents the execution of a 5-version system. When one of the versions successfully completes its execution, a token is deposited in count, causing the next in the set of transitions labelled *correct* to fire. When the *timeout* completes, a vote is taken, and a token deposited into one of the *failure* or *success* places.

Although timed transitions are used, the model of Geist *et al.* does not provide detailed timing information. The success/failure times of the individual alternates are hidden within the voting mechanism, and only the reliability distribution is provided as output. In their paper, Geist *et al.* derive *mean time to failure* information based on the average number of executions of the entire N-version programming system before the *failure* state is entered. The model of Geist *et al.* is, therefore, seen to capture some of the timing properties of an N-version programming system, but in the final analysis this detailed timing information is discarded, and coarse-grained metrics provided instead.

3.2.6.2 Shieh et al.

The work of Shieh *et al.* [83] models the process of rollback recovery and checkpointing using a stochastic Petri net. In this model, two basic primitives are derived: the *fault transition unit*, FTU, which manages the rollback of a process, and communicates this information to other communicating processes, causing them to stop executing. The *synchronisation transition unit*, STU, enables a non-local stopped process to rollback its execution to its most recent checkpoint. Both the FTU and STU are derived in terms of stochastic Petri net fragments.

A number of additional places and transitions are generated algorithmically, to represent the operation of a system, and these are linked with FTU and STU blocks. The final result is a somewhat complex system, which has been generated semi-automatically.

Metrics derived from the model of Shieh *et al.* include the average cycle time for a set of interacting processes; that is, the average time taken to perform a certain operation, in the presence of failure and rollback recovery. In addition, the average number of rollback attempts made may also be derived.

Like the work of Geist *et al.* discussed previously, the model of Shieh *et al.* does not provide detailed timing information, regarding the operation of a system. Timed transitions are used within the model to illustrate various aspects of a system's behaviour, but only high-level metrics are derived from this.

3.2.6.3 Hein & Goswami

The work of Hein & Goswami [37] provides an interesting example of the application of Petri net modelling combined with an event driven simulation. Named *Conjoint Simulation*, this technique splits a system model into two portions: a *failure-repair model* and an *architecture workload model.*

The failure-repair model is a timed Petri net model which controls and triggers the injection of errors into the components of the architecture workload model, and the fault-tolerant mechanisms built into that model. The marking of the Petri net representing the failure-repair model represents the state of failure, recovery and repair processes within the architecture workload model.

The architecture workload model depicts the structure of the system, such as the processors, communication links, I/O components, memory, etc; together with an event driven process model to simulate the workload on that hardware architecture.

The two models interact: events in the architecture workload model affect the firing time of transitions in the failure-repair model; the marking of the failure-repair model may cause processes in the architecture workload model to start/stop, or may roll-back operation of the architecture workload model to a predefined checkpoint, for example.

These models combine with the failure-repair model, based on a timed Petri net, providing synchronisation, timing, and precedence constraints; and the architecture workload model representing high-level system operation.

It is seen that whilst the timing properties of the Petri net model are used in this system, its main application is in providing control of concurrency and synchronisation effects. The majority of system timing functions are provided in the event driven architecture workload model, with the Petri net simply simulating fixed error rates.

3.3 Discussion

The discussion in section 3.1 focused on Markov models for the behaviour of a real-time system. A number of examples were presented, illustrating several important classes of Markov model:

- High-level functional models which classify the behaviour of a system into a small number of states, and model the transitions between those states. Such models allow for some estimation of system failure probabilities, but neglect timing properties entirely.
- It has been noted that such high-level models may not be sufficient, in themselves, for modelling the transitions between differing program blocks, since the memoryless assumption of the Markov model is too restrictive. This restriction may be removed in certain cases, but this typically leads to state-space explosion, and may result in models for which solution is computationally infeasible.

These results do not dismiss Markovian modelling techniques: they do, however, restrict their use to certain classes of system, and indicate that care must be taken if a Markov model of a system is not to become too complex to solve.

The discussion in section 3.2 focused on Petri net models. The basic Petri net system has different emphasis when compared with a Markovian model: it enables structural and reachability analysis, but is poor when probabilistic and timing models are required. A number of extended classes of Petri net model are discussed, which overcome these limitations in a number of different ways, but the underlying solution method for the performability of these systems is still Markovian. It is, however, clear that the added structure imposed on the underlying Markov chain by the higher level Petri net model is beneficial: the state space may be large, but it is typically regular, and numerical evaluation is often subject to certain optimisation techniques which permit efficient solution of these models.

It is of great importance to be able to derive the probability distribution of process completion times, in order to have some means of developing an execution schedule to meet all required deadlines, even in the presence of failures and error recovery. Further, it is clear that, despite the power of the underlying modelling techniques, the models developed to date are insufficient for modelling certain important properties of real-time systems. In particular, there has been little work which attempts to derive realistic measures of the timing properties of such systems: the models surveyed either ignore timing completely, or discuss only coarse measures such as mean completion time.

This illustrates a problem with current approaches to modelling real-time systems. It is usual for the timing properties of the system to be abstracted away so as to give each process a maximum execution time. Provided such a maximum time can be assigned, it is then possible to devise scheduling algorithms which, given sufficient resources, will ensure that all deadlines are met. These algorithms are pessimistic since they rely on the upper bound of a process' execution time, whereas in real systems, the probability of errors occurring is low and the execution time of most processes is typically much less than the maximum. The system therefore operates with much slack-time, implying low efficiency but high reliability. If the probability distribution of the process' execution times is known, it should be possible to design a system which relies on this to attain much improved efficiency, whilst still managing to operate within a tolerable level of risk.

The remainder of this thesis will introduce a technique which enables the probability distribution of process execution times to be determined, leading to improved scheduling techniques.

Chapter 4

A New System Reliability Model

As was discussed in chapter 3, there exist many techniques for modelling the reliability and performance of real-time systems. It was noted that, despite the power of these techniques, there has been little work conducted to determine the timing properties of real-time fault-tolerant systems. In this chapter, a new reliability model is developed which provides a framework for the development such techniques.

This chapter is divided as follows: Firstly, section 4.1 discusses the underlying assumptions upon which this model is based. This is followed, in sections 4.2 and 4.3, by a formal definition of this model. Finally, section 4.4 summarises the chapter.

4.1 Background

A number of experimental studies have been conducted into the failure characteristics of software systems [46, 65]. These studies, together with theoretical results such as those presented in [33, 48, 50, 53, 64] indicate that it is possible to achieve an accurate prediction of the failure characteristics of a software system using very simple models, and indeed, it has often been proposed that a random-fault model will suffice. Such a model is of use because of its ease of application and similarity to hardware reliability models, allowing similar techniques to be applied to the modelling of both hardware and software.

The notion of software faults occurring randomly is not intuitively obvious. In particular, a software system may unwisely be thought of as being purely deterministic – given a specific set of inputs a certain output will arise, and that same output will arise whenever that set of inputs is presented to the system. How can such a system conform to a random-fault model?

The simple answer is, of course, that it cannot. However, it can be seen that although the underlying faults do not occur randomly, their manifestation can appear to follow the random-fault model. A typical embedded control system will comprise a set of interacting software processes, together with a number of hardware devices. Interactions occur not only between software processes, but also between software processes and hardware devices and between hardware devices. In addition, interactions may occur between different parts of the system due to the flow of information through the external environment. This is illustrated in figure 4.1.

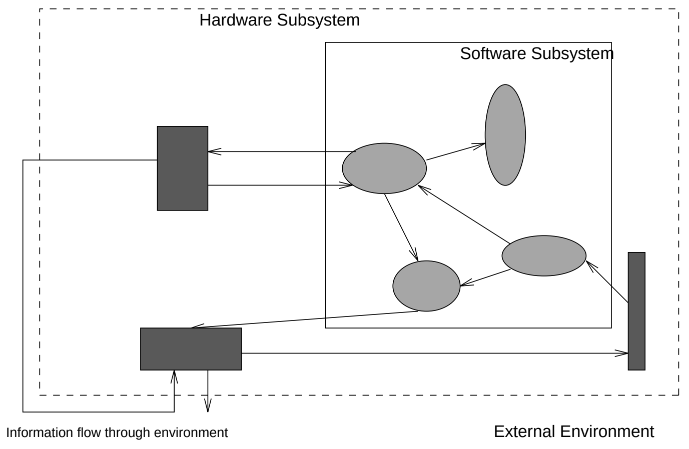

Figure 4.1: The structure of a typical embedded control system

The software comprising an embedded system such as this will have a large input space: It is directly affected by software-software interactions and software-hardware interactions and also indirectly affected by hardware-hardware interactions and the influence of the external environment. As the number of inputs to the system increases, and more and more external devices are included, it becomes increasingly difficult to determine the system boundary and the number of possible interactions increases rapidly. Furthermore, it is typical that embedded systems have a temporal dimension to their input space: Identical inputs may well produce different outputs at different times.

It can readily be seen that, for all but the simplest of systems, the input space is so large, and the interactions which occur are so subtle and complex, that it is effectively impossible to predict the path a system will take through its input space.

This conclusion is supported both by theoretical work, such as that of Littlewood [53], and by experimental data [32, 66]. In addition, Laprie & Kanoun [50] note that:

> "In the case of software, the randomness comes at least from the trajectory in the input space which will activate the faults. In addition, it is now known that most of the software faults which are still present in operation, after validation, are "soft" faults, in the sense that their activation conditions are extremely difficult to reproduce; hence the difficulty of diagnosing and removing them, which adds to the randomness."

As an example of this, a study of the error logs from a number of Tandem systems [32] concluded that only one fault out of 132 was not a soft fault.

From the above arguments, it seems reasonable to model a system's path through its inputs as a random-walk in a multi-dimensional space. This is transformed by the system to provide a path through the output space which is necessarily also modelled as a random walk. This is illustrated in figure 4.2.

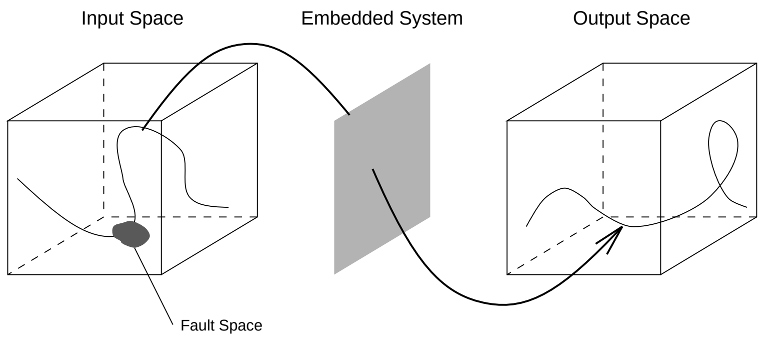

Figure 4.2: Random walk through a system's input space

There are typically a number of points in the input space which will give rise to faults in the system, and those faults may, eventually, cause errors to manifest themselves. Such errors, if untreated, may cause system failures. Due to the semi-random nature of a system's progress through its input space, and the complex mapping from input to output, it is noted that faults which are close in the input space will not necessarily give rise to faults which are close in the output space.

This then is the basis for the system model to be developed here. It is assumed that the system's input space is sufficiently large, and the tasks to be undertaken sufficiently complex, that a random-fault model such as this is applicable. It is considered that such an assumption is not unrealistic, indeed it is the basis for a number of other models [50, 53, 64], and certain experimental data [32, 65] have been collected which appear to confirm the validity of this approach. The work conducted by Laprie [48] also lends support to this, when it is noted that

> "...the constancy of the hazard rates, although it is an *a priori* unrealistic hypothesis, turns out to be satisfactory."

It is further noted that a study made of the reliability logs of Tandem systems [32] provides evidence for this claim, as indeed does the work of Musa at Bell Labs [64, 65], and that of the European Space Agency [29], where it is noted that

> "Software failure is a process that appears to the observer to be random, therefore the term reliability is meaningful when applied to a system which includes software and the process can be modelled as stochastic."

It can therefore be seen that the random-fault model as applied to software and combined hardware-software systems provides a reasonable fit with experimental data with a relatively simple theoretical background.

4.2 Underlying Mathematical Model

The techniques discussed in chapter 3 provide a number of ways by which the reliability and performance of real-time fault-tolerant systems may be modelled. Despite the power of these techniques, little work has been undertaken to determine the timing properties of such systems. There are a number of reasons for this: pure Markovian and Petri net models lack the descriptive power necessary to conveniently describe the problem domain, whilst the extended Petri net models provide, perhaps, excessive power and some simplification is desirable.

In this section a new mathematical model is described. This is a stochastic model, derived primarily from Markov chain theory, with modifications to allow for simple process interactions. The underlying network model borrows a number of concepts from Petri net theory, not least the notation used. Despite the notational similarities, however, this is primarily a Markov model, not a Petri net system.

The basis of this new model is a multi-graph consisting of a set of *places* and a set of *transitions* connected by directed arcs. The system state is defined by the probabilistic distribution of a set of tokens amongst these places. Changes in the system state are indicated by movement of tokens along the arcs, from place to place by means of the intermediate transitions. All tokens move at once, in step-time. A transition cannot fire until it has a token in each of its input places, and when it does fire it sends a single token to one of its output places, determined probabilistically by the *transition probabilities* labelled on the arcs leading away from the transitions. A place may have multiple input arcs and hence, may receive multiple tokens. A place will output a token down each of its output arcs. A place will therefore create or destroy tokens as required.

The precise details of the transition probabilities will be described in section 4.3. At present, it is sufficient to describe the basic modes of operation of the model (figure 4.3). The notation borrows heavily from Petri net models: circles represent places, bars represent transitions, and arrows represent connecting arcs.

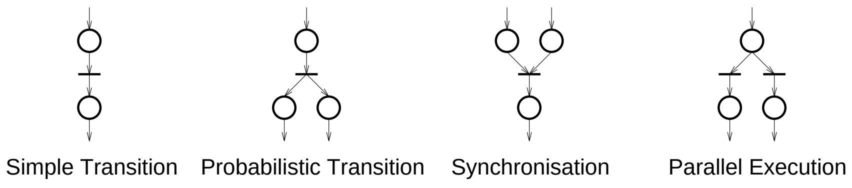

Figure 4.3: Basic modes of execution

The *simple transition* and *probabilistic transition* modes correspond to a standard Markov chain model: Tokens are neither created or destroyed. In these modes the system shows multiple possible paths of execution — it can perform *one* action from a choice of many possibilities — this is modelled as a *transition* with multiple output arcs.

A system which permits concurrent execution of multiple paths is also possible, and is modelled by a *place* with multiple output arcs. This is the parallel execution mode, and shows token creation: a single token is received as input, and a token emitted down *each* output arc. Further, it is possible to model synchronisation among these concurrent processes by means of transitions with multiple input arcs. Such transitions cannot fire until all their input places contain tokens, and so they introduce synchronisation into the execution of the system, and destroy excess tokens.

Further, more advanced modes of operation, allow for multi-step transitions, which take several time steps to execute. This is a simplification to reduce the run-time of the model: the size of the state-space is reduced, resulting in faster numeric solution of the model.

This model is, therefore, seen to include a number of desirable features taken from extended Petri net techniques: timed transitions, synchronisation, and easy expression of concurrency. The underlying basis is Markovian analysis, allowing performance analysis to be undertaken. A formal definition of this model is now necessary, and is provided in section 4.3.

4.3 Formal definition of the model

4.3.1 Basic Network Model

The basis of the model is a set of places with probabilistic movement of tokens between them. This is defined by a four-tuple

$$C = (\Theta, \Lambda, I, O) \tag{4.1}$$

where

- $\Theta = \{\theta_1, \theta_2, \dots, \theta_N\}$ is a finite set of *places* with $N \geq 2$ representing the system state.
- $\Lambda = \{\lambda_1, \lambda_2, \dots, \lambda_M\}$ is a finite set of *transitions* with $M \geq 1$, representing the possible movements between states.
- The *input function*, I, and the *output function*, O, define the following mappings between Θ and Λ:
  - Transition to place:
  $$I : \Lambda \mapsto \Theta \tag{4.2}$$
  $$O : \Lambda \mapsto \Theta \tag{4.3}$$
  - Place to transition:
  $$I : \Theta \mapsto \Lambda \tag{4.4}$$
  $$O : \Theta \mapsto \Lambda \tag{4.5}$$
  - Transition to transition:
  $$I : \Lambda \mapsto \Lambda \tag{4.6}$$
  $$O : \Lambda \mapsto \Lambda \tag{4.7}$$

  It is noted that no mapping is defined for $\Theta \mapsto \Theta$, that would imply an instantaneous jump in the system state and is meaningless: such a combined place is always equivalent to a single place.

  The input and output functions, I and O, define arcs connecting the places and transitions of the network. These arcs are weighted. All arcs have weight $w = 1.0$, with the exception of the arcs defined by $O : \Lambda \mapsto \Theta$ and $O : \Lambda \mapsto \Lambda$ which together define the transition probabilities $T_{i,j}^{(n)}$ (see below), and have weight w : $0 \leq w \leq 1$. It is noted that the sum of the arc-weights for arcs leaving any transition must be unity. Multiple arcs may leave each place, resulting in token creation, since the sum of these probabilities will be greater than unity. This permits the simple modelling of concurrency.

The following restrictions are made

- The set of places, Θ, and the set of transitions, Λ, are disjoint:

$$\Theta \cap \Lambda = \emptyset \tag{4.8}$$

- Two places θ_i and θ_j may be connected by at most *one* single-step transition:

$$|O(\theta_i) \cap I(\theta_j)| \leq 1 \tag{4.9}$$

  Multiple connecting transitions can always be combined into a single transition with the combined probabilities, and hence are unnecessary.

- A transition may take input from a set of places *or* a set of transitions, but not both:

$$I(\lambda_i) \cap \Theta \neq \emptyset \implies I(\lambda_i) \cap \Lambda = \emptyset \tag{4.10}$$

$$I(\lambda_i) \cap \Lambda \neq \emptyset \implies I(\lambda_i) \cap \Theta = \emptyset \tag{4.11}$$

  This allows for n-step, timed, transitions, but prohibits a transition which is in the process of firing from being delayed by the state of other portions of the system. Timed transitions are simply an optimisation, to eliminate unnecessary places; they have a fixed firing time. For the problems to which this model is to be applied, transitions with probabilistic firing delays are not useful.

- A transition can take input from at most one other transition:

$$|I(\lambda_i) \cap \Lambda| \leq 1 \tag{4.12}$$

- A transition which has output to one or more other transitions can have at most one input:

$$\forall \lambda_k : O(\lambda_k) \cap \Lambda \neq \emptyset, |I(\lambda_k)| \leq 1 \tag{4.13}$$

- A transition with input arcs from multiple places represents a synchronisation primitive; the set of such transitions is $\Lambda_{sync} \subset \Lambda$. A transition in this set, $\lambda_{sync} \in \Lambda_{sync}$, is described by:

$$I(\lambda_{sync}) \cap \Lambda = \emptyset \tag{4.14}$$

$$|I(\lambda_{sync}) \cap \Theta| \geq 2 \tag{4.15}$$

A place, θ_s, which has output to such a transition is restricted to having only that single output:

$$\theta_s \in I(\lambda_{sync}) \implies O(\theta_s) = \lambda_{sync} \tag{4.16}$$

If multiple outputs were allowed from such a place, synchronisation would not be possible: a "concurrent-fork-and-join" primitive is not useful.

These definitions provide the basic system structure.

4.3.2 Single-step execution rules

The *time-independent single-step transition probability* between places θ_i and θ_j is denoted by $T^{(1)}_{i,j}$. This is the probability that a movement can occur from place θ_i to place θ_j provided that there is a single-transition, λ_k, linking these two places. It can be seen that $T^{(1)}_{i,j}$ is the product of the weights of the arcs linking places Θ_i and Θ_j, via the intermediate transition, λ_k.

In order for a single transition, λ_k, to link two places, that transition must be an element of the set of output transitions of one of the places, and an element of the set of input transitions of the other place:

$$\lambda_k = O(\theta_i) \cap I(\theta_j) \tag{4.17}$$

If $\lambda_k = \emptyset$ then no single-step transition is possible between states θ_i and θ_j. However, if $\lambda_k \neq \emptyset$ then a single-step transition is possible, and the set λ_k holds the transition by which that movement is made.

It is now necessary to define the *time-dependent single-step transition probability* between places θ_i and θ_j at time t. This is the probability that a single-step transition will occur, based around the system state at a specified time. It is not possible for a transition to fire until all its input places are enabled; and a place is said to be enabled if there is a non-zero marking for that place. Hence, the time-dependent single-step transition probability is defined as

$$\pi^{(t)}_{i,j} = T^{(1)}_{i,j} \prod_{I(\lambda_k)-\theta_i} P_k(t) \tag{4.18}$$

where λ_k is defined in equation 4.17 and $P_k(t)$ is the marking for place θ_k at time t, defined by equation 4.22. The time-independent single-step transition probability, $T^{(1)}_{i,j}$, is multiplied by the product of the probabilities that each of the input places to that transition are enabled, with the exception of the input place from which the transition is made. This exception is made for two reasons: Firstly, if the place θ_i is not included, then a system with only single input arcs becomes equivalent to a simple Markov chain model. Second, if the input from place θ_i is included then the definitions required by the model become mutually recursive and are impossible to evaluate.

It is noted, once again, that arcs leading from place to transition have unity weight. The transition probability is determined by arcs leading from transition to place.

4.3.3 n-step execution rules

In section 4.3.1 it was specified that movement can occur between two transitions, λ_i and λ_j, subject to certain restrictions on topology. This allows time-independent n-step transitions between places to be described. These transition probabilities are denoted by $T^{(n)}_{i,j}$ and indicate a movement from place θ_i to place θ_j which passes through n transitions, where $n > 1$, and which *does not* pass through any intermediate places. As for the single-step transition probabilities, $T^{(1)}_{i,j}$, these n-step transition probabilities are formed by the product of the weights of the arcs traversed. Given this definition, and the definitions of section 4.3.2, it is possible to derive an expression for the *time-dependent* n-step transition probability, $p^{(n,t)}_{i,j}$, between places θ_i and θ_j at time t.

It is noted that the n-step transition probabilities *for a Markov chain* are given by equations 3.2 and 3.3 on page 19. It is noted that this probability is *time-independent*, and allows only n-step movements which pass through other intermediate places, since Markov chains do not allow for mappings $\Lambda \mapsto \Lambda$.

This definition can be extended by allowing n-step movements which use only transitions, $T^{(n)}_{i,j}$, although it is *not* possible to simply add $T^{(n)}_{i,j}$ to the above equation, since there may be other indirect paths by which a movement may occur, consisting of an m step transition-only movement, and an $(n-m)$ step movement using intermediate places. This, therefore, leads to the following expression for the n-step transition probabilities:

$$p^{(n)}_{i,k} = \sum_{j=1}^{N} p_{i,j} p^{(n-1)}_{j,k} + \sum_{m=2}^{n} \sum_{j=1}^{N} T^{(m)}_{i,j} p^{(n-m)}_{j,k} \tag{4.19}$$

where $p^{(0)}_{j,k}$ is defined as in equation 3.3, and N is the number of places. This consists of the transition probability as if the direct multi-step transitions were not present, specified by the first summation term, with the addition of the probability of making the transition by any combination of direct, $T^{(m)}_{i,j}$, and indirect, $p^{(n-m)}_{j,k}$, routes.

It is then a simple matter to add timing information to this; The time independent single step transition probability, $p_{i,j}$, is replaced by the time-dependent probability, $\pi_{i,j}^{(t)}$ (see equation 4.18), and a timing parameter is added into the definition of the n-step transition probability, $p_{i,k}^{(n)}$. This provides an expression for the time-dependent n-step transition probability as follows:

$$p_{i,k}^{(n,t)} = \sum_{j=1}^{N} \pi_{i,j}^{(t-n)} p_{j,k}^{(n-1,t)} + \sum_{m=2}^{n} \sum_{j=1}^{N} T_{i,j}^{(m)} p_{j,k}^{(n-m,t)} \tag{4.20}$$

where

$$p_{j,k}^{(0,t)} = \begin{cases} 0 \text{ if } j \neq k \\ 1 \text{ if } j = k \end{cases} \tag{4.21}$$

4.3.4 Markings and system state

The system marking function is defined by the absolute probability distribution, $P_i(t)$, representing the probability that there is at least one token in place θ_i at time t. This definition is based around the equivalent definition for a Markov chain system, defined by equation 3.5. A number of modifications must, however, be made to this definition:

- Time-dependent transition probabilities, as described in section 4.3.2, must be allowed.
- Additions must be made to allow a place to maintain it's marking to allow synchronisation.

The first modification may be made by simply substituting the time dependent n-step transition probability into equation 3.5. The second modification is made by the addition of a constant, ρ, to represent held-over marking. This leads to a definition for the absolute probability as in equation 4.22:

$$P_i(t) = \rho + \sum_{j=1}^{N} P_j(0) p_{j,i}^{(t,t)} \tag{4.22}$$

where

$$\rho = \begin{cases} P_i(t-1) & \text{if } \exists\ \lambda_{sync} \in \Lambda_{sync}\ :\ \theta_i \in I(\lambda_{sync}) \\ & \text{and } \exists\ \theta_s \in I(\lambda_{sync}) - \theta_i\ :\ P_s(t-1) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.23}$$

where $P_i(0)$ denotes the initial probability distribution for the system.

The constant, ρ, in equation 4.22 is interesting. If a place has output to a synchronisation transition, λ_{sync}, then that place will retain its marking until *all* places leading into that transition have non-zero marking. Since the definition for single step execution rules, equation 4.18, is such that such synchronisation transitions will not fire until all their input places are enabled, this conspires to introduce synchronisation into the model.

The marking is hence a vector which changes with time, based upon the execution rules of the system, and is therefore a representation of the system state at a particular time. By sampling the marking at various states, the execution of a system may be modelled.

4.4 Discussion

The model developed in this chapter has a number of similarities to the techniques discussed in chapter 3, but also a number of important differences. The basis of the model described here is a Markov model, which allows for simple performance evaluation. This is extended with two concepts: concurrent execution, and synchronisation. Together, these additions allow a greater expressiveness than is possible in simple Markovian models. The notation used by Petri net models is borrowed, since this allows these concepts to be explicitly indicated. Indeed, it can be seen that this model is similar in many ways to some classes of extended Petri net model: it does not, however, require the full generality of these extended models.

It is felt, therefore, that the model developed here has a number of practical advantages over both Markov chain models and extended Petri net systems, namely:

- It permits easy modelling of concurrency and synchronisation.
- Performance evaluation is possible, since the underlying Markov model allows marking probabilities to be calculated.
- Since the model is more restricted than, say, stochastic Petri nets; it permits simpler solution. The uses envisaged for this model, as will be discussed in chapters 5 and 6, result in relatively small, regular, Markov chains; and efficient solution is possible.

These features make this model easy to use for the purposes of modelling the failure time distribution of real-time systems subject to faults.

It is noted that both simple Markov chain models and some classes of extended Petri net model are capable of modelling systems in this manner, but it is felt that this model is more suited to this particular task. That is, this new model provides for simpler modelling of this class of fault-tolerant systems under investigation, when compared with the Petri net and Markov chain models. This will be discussed further in later chapters of this thesis.

This concludes the discussion of the basic modelling technique which, in chapters 5 and 6, will be applied to the problem of modelling the timing properties of real-time systems subject to failures.

Chapter 5

Generic Real Time System Model

The mathematical framework described in chapter 4 provides the basis upon which system reliability models may be developed. In this chapter, that framework is used to derive a new model for the behaviour of real-time systems. This is a discrete time model with a lattice structure which models the progress of a computation from its initial state to one of several final states: completed, detectable fault, hidden fault and failed. This model allows for both the functional and temporal behaviour of a system to be represented in a single high-level model. It is derived from generic properties of real-time systems, hence being independent of any specific design/implementation technique for such systems. This is an improvement on traditional system reliability models which typically focus on functional correctness and do not adequately model the temporal properties of such systems.

This chapter is structured as follows: section 5.1 provides a definition of this model building from the generic, high-level, properties of real-time systems and the mathematical framework described in chapter 4. Section 5.2 discusses the various system parameters needed by this model, and how they may be estimated. Section 5.3 describes a number of testbed systems, the behaviour of which is studied in section 5.4 leading to an understanding of the basic properties of this model. Finally, section 5.5 summarises the chapter.

5.1 A Generic Real Time System Model

In order to model the behaviour of a real-time system, certain properties of that system must be known. In particular, the execution time bounds of the system should be known, together with the probability that the system may exceed those bounds. Given this information, and in the absence of faults, such a system may be modelled as a simple state chain with probabilistic transition to a *completed* state which can only occur during a specified time period (figure 5.1). The transition probabilities to the completed state must match the completion profile, that is, the plot of completion probability against time, for the system in the absence of failures.

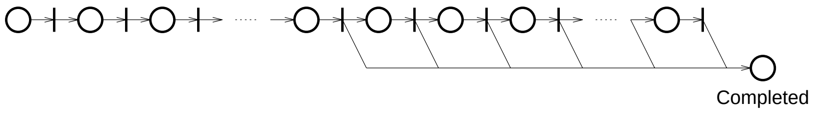

Figure 5.1: Basic State Chain

The model illustrated in figure 5.1 is, of course, overly simplistic and must be extended in order to account for the presence of faults within the system. For example, Pucci [77] notes that

> "An important distinction is between errors whose manifestation is identified in the system (*detected errors*) and errors whose manifestations is not (*undetected errors*)."

In particular, the effects of hidden faults are noted:

> "Undetected errors are the most insidious ones because of their effect on the future behaviour of the software. The delivered incorrect results can spread [...] to the entire software system and create state inconsistencies. In contrast, the system will be aware of the occurrence of a detected error and can attempt recovery actions."

It is therefore of great importance that the model developed here allows for the effects of both detectable and hidden faults.

The first class of fault may be modelled by the addition of a *detectable fault* state to the model describing the system. A transition is made from each state in the basic state chain to this state, with probability determined using a random fault model (figure 5.2). Since the system obeys a random-fault model, as discussed in chapter 4, the transition probability for each of these paths is uniform. It is noted that faults which would cause a time over-run fall into the detectable category, and so there is no need to further model a process which can exceed its time bounds.

The second class of fault leads to a more complex model, requiring a parallel state chain to represent a system which is still functioning, but with a hidden fault [77]. These states mimic the function of the original state chain, and lead to the *hidden fault* and *failed* states (figure 5.3). The system is therefore partitioned into two state chains: progress of the system through its computation is indicated by transitions along a state chain, and changes in the operational mode of the system (whether the occurrence of failure, or recovery from that failure) are indicated by transitions between the two state chains.

The transition probabilities for this parallel set of states mirror those of the original, fault-free, states: that is, the transition probabilities into the *hidden fault* state equal those for transitions into the *completed* state, and the transition probabilities into the *failed* state equal those for the *detectable fault* state. An example of this is shown in figure 5.4, from which it is noted that:

$$T^{(1)}_{x,22} = T^{(1)}_{x+20,41} \text{ for } x \in \{9, 11, 13, 15, 17, 19\} \tag{5.1}$$

$$T^{(1)}_{x,21} = T^{(1)}_{x+20,42} \text{ for } x \in \{3, 5, 7, 9, 11, 13, 15, 17, 19\} \tag{5.2}$$

The transitions between the two parallel state chains have uniform probability, according to the random-fault model. The two state chains represent the system executing the same algorithm; differences in the behaviour between the two state chains are due entirely to the differences in the state space upon which they act. Since the behaviour of the algorithm for all possible input sets is approximated in the arc weights for transitions to the completed and hidden fault states, it is assumed that the behaviour caused by this class of fault is implicitly included in these probabilities.

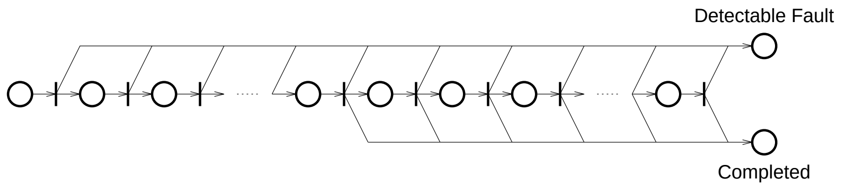

Figure 5.2: System model with detectable faults

This then leads to the final definition of the model, comprising two parallel state chains representing normal execution and execution with a hidden fault. These are interconnected with a lattice structure which models hidden fault occurrence and recovery. This model may then be subjected to analysis as described in section 4.3, leading to the determination of the marking function (Equation 4.22) for the four final states of this model. It is this marking function which represents the system behaviour with time.

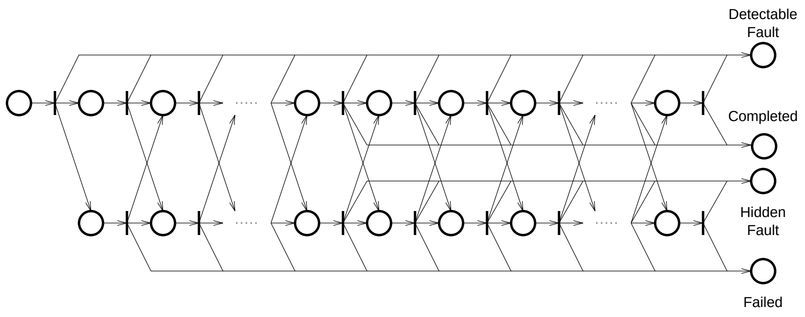

Figure 5.3: System model with hidden faults

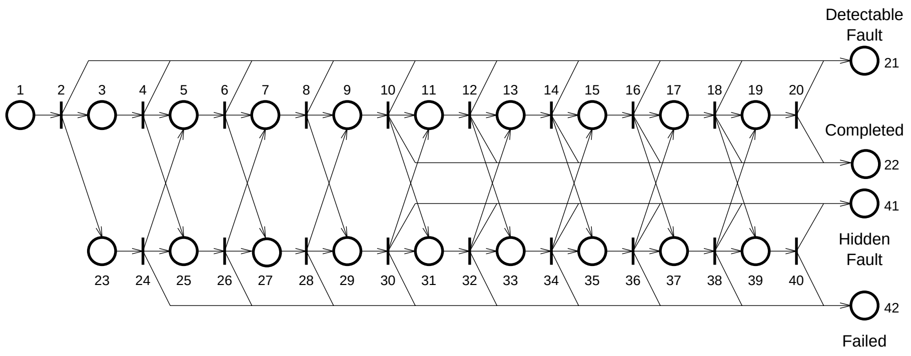

Figure 5.4: System model with hidden faults (example)

This model can then be used to determine *both* the functional and the temporal correctness of a system. The *functional* correctness is indicated by the probability distribution of the system between the four final states of the model: completed, detectable fault, hidden fault, and failed. The *temporal* correctness is indicated by plotting the timing profile to show the distribution of these probabilities with respect to time.

5.2 Parameter Estimation

Implicit in the discussion so far has been the precise nature of the transition probabilities, $T^{(1)}_{i,j}$, of the lattice model. These are divided these into four categories:

- *Probability of completion,* p_c: This is the probability that the system completes execution at any given time step; independent of the occurrence of faults. This value must be derived from knowledge of the algorithm used by the process and/or test data. This is the transition probability for the arcs leading to the *completed* and *hidden fault* states. In figure 5.4 these are the transitions to states 22 and 41 respectively. It is noted that the arc weights cannot be directly observed: rather the probability that the system completes at a particular time may be estimated, and from this the arc weights derived, as discussed below.

- *Probability of detectable fault,* p_d: This is the probability that the system fails in such a manner that the failure can be detected before the normal completion time of the system. The probability may be estimated from test data, or from experience with similar systems. This is the transition probability for arcs leading to the *detectable fault* and *failed* states. In figure 5.4, these are the transitions to states 21 and 42 respectively.

- *Probability of hidden fault,* p_f: This is the probability that a fault occurs which does not give rise to an error detectable at run-time. Such a fault may be detected after completion of the process, and hence may be estimated based on the results of a system acceptability test. This probability, together with the probability of hidden recovery, defines the transition probabilities on the arcs interconnecting the two main state chains of the model.

- *Probability of hidden recovery,* p_r: This is the probability that the system recovers silently from a hidden (unobservable) fault. It may be estimated in a similar manner to the probability of a hidden fault.

With the exception of the completion probability, p_c, these transition probabilities are expected to be uniform, and to follow a random-fault model, as discussed in chapter 4.

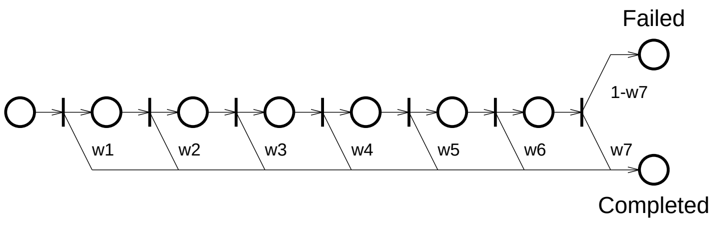

Figure 5.5: Derivation of arc weights from completion profile

It has been noted that it is not possible to directly observe the p_c arc weights. However, the completion probability of the system with respect to time may be measured, and given this information it is possible to deduce the arc weights which will produce those completion probabilities.

If it is assumed that faults do not occur, that is $p_d = p_f = p_r = 0$, the generic system model may be reduced to the simple form of figure 5.1. Taking a specific system, figure 5.5, as an example, it is possible to derive the completion profile of the system in terms of the arc weights, w_n, as shown in equation 5.3.

$$\begin{aligned}
P(\text{Completed}, t = 1) &= 0 \\
P(\text{Completed}, t = 2) &= w_1 \\
P(\text{Completed}, t = 3) &= (1 - w_1)w_2 \\
P(\text{Completed}, t = 4) &= (1 - w_1)(1 - w_2)w_3 \\
P(\text{Completed}, t = 5) &= (1 - w_1)(1 - w_2)(1 - w_3)w_4 \\
P(\text{Completed}, t = 6) &= (1 - w_1)(1 - w_2)(1 - w_3)(1 - w_4)w_5 \\
P(\text{Completed}, t = 7) &= (1 - w_1)(1 - w_2)(1 - w_3)(1 - w_4)(1 - w_5)w_6 \\
P(\text{Completed}, t = 8) &= (1 - w_1)(1 - w_2)(1 - w_3)(1 - w_4)(1 - w_5)(1 - w_6)w_7
\end{aligned} \tag{5.3}$$

These expressions may be inverted to express the arc weights in terms of the completion profile, as shown in equation 5.4.

$$\begin{aligned}
w_1 &= \text{P(Completed}, t = 2) \\
w_2 &= \frac{\text{P(Completed},t=3)}{(1-w_1)} \\
w_3 &= \frac{\text{P(Completed},t=4)}{(1-w_1)(1-w_2)} \\
w_4 &= \frac{\text{P(Completed},t=5)}{(1-w_1)(1-w_2)(1-w_3)} \\
w_5 &= \frac{\text{P(Completed},t=6)}{(1-w_1)(1-w_2)(1-w_3)(1-w_4)} \\
w_6 &= \frac{\text{P(Completed},t=7)}{(1-w_1)(1-w_2)(1-w_3)(1-w_4)(1-w_5)} \\
w_7 &= \frac{\text{P(Completed},t=8)}{(1-w_1)(1-w_2)(1-w_3)(1-w_4)(1-w_5)(1-w_6)}
\end{aligned} \tag{5.4}$$

From this, it is clear that a general expression for the arc weights must take the form shown in equation 5.5:

$$w_n = \frac{\text{P(Completed}, t = n + 1)}{\prod_{i=1}^{n-1}(1 - w_i)} \tag{5.5}$$

This enables the derivation of arc weights from completion profiles in the absence of faults, and equally the derivation of arc weights to match a desired completion profile, for the purposes of studying the behaviour of the model.

Of course, a real system will exhibit failures, and the parameters p_d, p_f and p_r will have non-zero value. In order to derive the p_c arc weights from the observed completion probabilities it is, therefore, necessary to factor out the effects of the other failure modes.

Study of the generic model illustrated in figure 5.3, together with the application of the mathematical framework described in section 4.3, allows a number of statements to be made linking the observable probability distribution of the four final states of the generic model with the model parameters.

Taking the example system illustrated in figure 5.4, it is clear that for any state, s, in the set $s = \{1, 3, 5, 7, 9, 11, 13, 15, 17, 19\}$ the following expressions hold:

$$\text{P(Completed}, t + 1) = p(s, t)\ p_c^n \tag{5.6}$$

$$\text{P(Detectable-Fault}, t + 1) = p(s, t)\ p_d \tag{5.7}$$

Combining equations 5.6 and 5.7 leads to equation 5.8 which shows a link between the system completion probabilities, p_c^n, and the probability of detectable fault, p_d.

$$\frac{P(\text{Completed}, t+1)}{P(\text{Detectable-Fault}, t+1)} = \frac{p_c^n}{p_d} \tag{5.8}$$

Given a formulation such as that in figure 5.8 it is possible to estimate p_d if it is assumed that the completion and detectable fault probabilities are observable at the final states of the model. If it is further assumed that p_c^n, that is the n-th completion probability, may take any value in the range $0 \ldots 1$, then the value of p_d may be bounded as shown in equation 5.9.

$$0 \leq p_d \leq \frac{P(\text{Detectable-Fault}, t+1)}{P(\text{Completed}, t+1)} \tag{5.9}$$

Back substitution of this bounded value for p_d into equation 5.8 allows for estimation of the completion probabilities, p_c^n.

The value of p_d and the completion probabilities, p_c^n, may be further bounded by application of similar techniques to the ratio of the observed probability distribution of the hidden fault and failed states of the model, as per equation 5.10:

$$\frac{P(\text{Hidden-Fault}, t+1)}{P(\text{Failed}, t+1)} = \frac{p_c^n}{p_d} \tag{5.10}$$

Once again, taking the example of figure 5.4, it is clear that for any state, n, in the set

$$n = \{1, 3, 5, 7, 9, 11, 13, 15, 17, 19\}$$

and any state, m, in the set

$$m = \{23, 25, 27, 29, 31, 33, 35, 37, 39\}$$

the following expressions hold:

$$P(n,t) = (1 - p_d - p_f - p_c^n)P(n-1, t-1) + p_r P(m-1, t-1) \tag{5.11}$$

$$P(m,t) = (1 - p_d - p_r - p_c^n)P(m-1, t-1) + p_f P(n-1, t-1) \tag{5.12}$$

Hence, it follows that

$$\frac{P(m,t)\ -\ p_f P(n-1,t-1)}{P(m-1,t-1)} + p_r = 1 - p_d - p_c^n \tag{5.13}$$

$$\frac{P(n,t)\ -\ p_r P(m-1,t-1)}{P(n-1,t-1)} + p_f = 1 - p_d - p_c^n \tag{5.14}$$

and therefore

$$\frac{P(m,t)\ -\ p_f P(n-1,t-1)}{P(m-1,t-1)} + p_r = \frac{P(n,t)\ -\ p_r P(m-1,t-1)}{P(n-1,t-1)} + p_f \tag{5.15}$$

Further, figure 5.4, shows that

$$P(n,t)p_d = P(\text{Detectable-Fault}, t+1) \tag{5.16}$$

$$P(m,t)p_d = P(\text{Failed}, t+1) \tag{5.17}$$

Substituting these results into equation 5.15 leads to equation 5.18:

$$\begin{aligned}\frac{P(\text{Failed}, t+1)\ -\ p_f P(\text{Detectable-Fault}, t)}{P(\text{Failed}, t)}\ +\ p_r \\ = \frac{P(\text{Detectable-Fault}, t+1)\ -\ p_r P(\text{Failed}, t)}{P(\text{Detectable-Fault}, t)}\ +\ p_f\end{aligned} \tag{5.18}$$

Since the failure and detectable fault probabilities are observable parameters of the systems, equation 5.18 allows for estimation of p_f and p_r, the probabilities of hidden fault occurrence and recovery.

For safety critical work, the upper bound on the values of the model parameters should typically be chosen, giving worst-case performance.

It is therefore seen that the parameters required by the model may be estimated based on test data from a real system. This model is therefore of use in a predictive role: given preliminary test data for a component it is possible to derive a reliability and timing prediction. A number of these may then be combined to predict the behaviour of an entire system.

5.3 Standard Completion Profiles

The generic system model developed in this chapter is the basis upon which a number of more advanced models may be built. In particular, reliability models for various fault tolerant system configurations will be derived from this base in chapter 6 of this thesis. It is therefore important that the behaviour of this generic model is well understood, and for this reason the behaviour of the model has been simulated under a number of different conditions.

The model requires a number of parameters to be specified before numeric simulation of the behaviour of a system can be performed. As discussed in section 5.2, three of these parameters (p_d, p_f and p_r) are simple uniform probabilities, and these values are determined based on reliability estimates for the system under consideration. The fourth parameter, p_c, represents the expected completion probability with respect to time for the system in the absence of faults, and consists of the arc weights for transitions to the *completed* state. To evaluate the effects of changes in these parameters, a standard system must be defined, with known parameters, and any comparison must be made relative to this standard system.

For the purpose of evaluating the behaviour of the model, three standard systems have been chosen: exponential, uniform and binomial. These three systems offer very different performance characteristics, and hence may be expected to provide a good comparison of the behaviour of the model.

To ease comparison, the three standard systems were chosen such that the overall completion probability, that is, the integral of the probability observed at the completed state of the model with respect to time, was in the range $(0.00001 \cdots 0.10000)$. These values were chosen to show the behaviour of the model under a broad spread of completion probabilities, they are not necessarily representative of any real application.

Further, all three systems were specified to execute for 50 time steps, with completion possible between time steps 30 and 50 inclusive. Once again, these values were chosen for the purpose of illustrating the behaviour of the model. Parameter estimation for the purposes of evaluating the behaviour of a real system is discussed in chapter 6.

These three standard systems are now discussed in turn.

5.3.1 Exponential Completion Profile

The exponential completion profile is the easiest to implement in the model, since it is the result of a system where all the p_c arc weights are uniform. By application of the techniques discussed in section 5.2 it can be seen that the completion profile for such a system may be defined as in equation 5.19 and illustrated by figure 5.6 (left) for a system with arc weight, $w = 0.1$.

$$P(\text{Completed}, t) = w(1 - w)^t \tag{5.19}$$

For a system where completion can only occur after an initial time, t_{ci}, it is necessary to modify equation 5.19, leading trivially to equation 5.20, and, for example, figure 5.6 (right) where $t_{ci} = 30$.

$$P(\text{Completed}, t) = \begin{cases} w(1 - w)^{t - t_{ci}} & \text{for } t_{ci} \leq t \leq t_{max} \\ 0 & \text{otherwise} \end{cases} \tag{5.20}$$

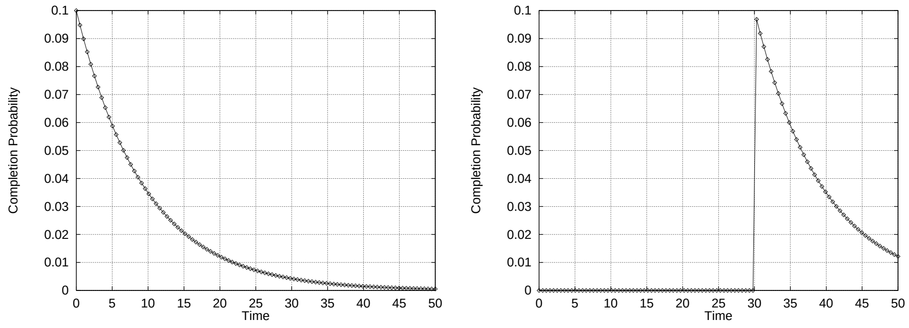

Figure 5.6: Exponential completion profile

Since this is a discreet time model, the area under these curves may be calculated by summing the data points, as shown in equation 5.21. This area is the overall completion probability for the system.

$$P(\text{Completed}) = \sum_{t=t_{ci}}^{t_{max}} w(1 - w)^{t - t_{ci}} \tag{5.21}$$

A plot of the overall completion probability for differing arc weights is shown in figure 5.7. This figure was generated with $t_{ci} = 30$ and $t_{max} = 50$.

It is a simple matter to solve equation 5.21 to find the arc weights, w, which give a specific value for the overall completion probability: these arc weights are shown in table 5.1 (left). For simplicity, these weights have been rounded to the values shown in table 5.1 (right), and it is these which were used to generate the standard exponential completion profiles (figure 5.8), and hence these are the values used for the parameter p_c in the exponential system[1].

[1]For compatibility with the other standard systems, the exponential system has been chosen with a

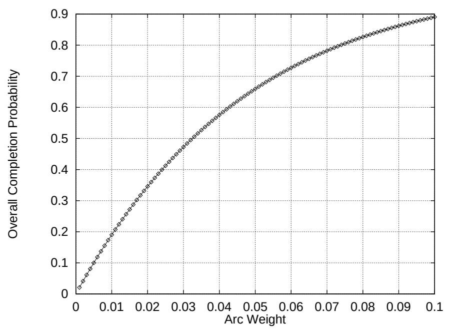

Figure 5.7: Overall completion probability: Exponential system

Overall Completion Prob.	Arc Weight	Overall Completion Prob.	Arc Weight
0.10000	0.005004600000	0.0999126000	0.0050000
0.01000	0.000478473000	0.0104477000	0.0005000
0.00100	0.000047641700	0.0010494800	0.0000500
0.00010	0.000004762130	0.0001049950	0.0000050
0.00001	0.000000476193	0.0000104999	0.0000005

Table 5.1: Arc weights for the exponential system

5.3.2 Uniform Completion Profile

For the uniform, or rectangular, distribution the area under the completion probability *vs.* time curve must take one of a fixed set of values. Since the width of the completion region is also fixed, the height of the curve is simply defined. These desired completion probabilities are illustrated in figure 5.9, left.

By application of the techniques described in section 5.2, it is possible to derive the arc weights, p_c, for the uniform completion profile systems: these are illustrated in figure 5.9, right, which shows the variation in arc weight through the system for arcs leading to the completed state. It can be seen that the curve for area 0.10000 shows increasing arc weight: this is in fact common to all five curves, although difficult to see in this figure. Such an increasing arc weight is expected in order to retain a constant completion probability as the proportion of systems executing decreases with respect to time, due to earlier completions.

small completion probability. Unfortunately, this leads to systems with a slow rate of decay, and the exponential nature of the system is not readily apparent, the decay becomes more visible for systems where is arc weight is of the order of 0.2 or above.

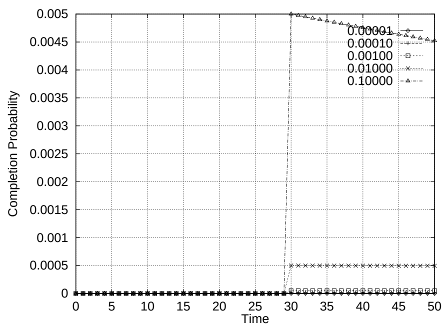

Figure 5.8: Standard exponential completion profiles

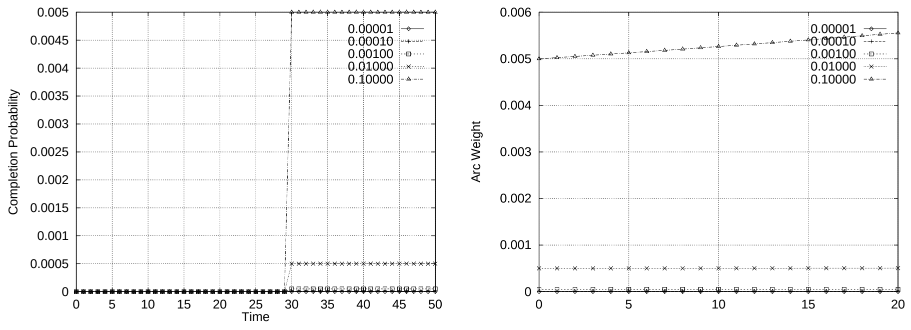

Figure 5.9: Standard uniform completion profiles, and corresponding arc weights

5.3.3 Binomial Completion Profile

A random variable, r, which has a binomial distribution will represent the total number of successes obtained in n repetitions of an experiment with probability, p, of success, this is defined by equation 5.22 [40, 42].

$$P(r) = \binom{n}{r} p^r (1-p)^{n-r} \tag{5.22}$$

For the purposes of this work, n and p will be regarded simply as parameters to be adjusted to produce a curve of the required shape, and their interpretation in terms of probability theory will be neglected.

A binomial distribution produces a "bell shaped" curve where the parameter n denotes the width of the curve and p locates the mean. For $p = 0.5$ the mean will be midway between the upper and lower bounds of the curve, if $p < 0.5$ the curve will be skewed towards the lower bound, and for $p > 0.5$ the curve is skewed towards the upper bound. This is illustrated in figure 5.10.

In order to match the other standard distributions, curves with a width of 20, and with

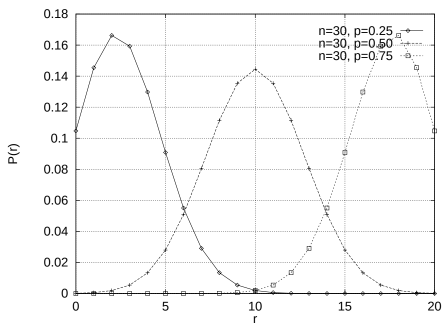

Figure 5.10: Binomial distribution: Effects of changing p

area in the range $\{0.10000, 0.01000, 0.00100, 0.00010, 0.00001\}$ are required. These are formed by taking the central 20 points of a binomial distribution with $p = 0.5$, and n selected to produce the required area. The values of n chosen are shown in table 5.2 together with the required and actual area under the curves. These truncated binomial distributions were then shifted from the range $0 \cdots 20$ to the range $30 \cdots 50$, to produce the final completion profiles shown in figure 5.11 (left). The arc weights required to produce such distributions are shown in figure 5.11 (right), these were calculated as described in section 5.2.

n	Required area	Actual area
162	0.10000	0.098661
68	0.01000	0.010338
42	0.00100	0.000941
32	0.00010	0.000113
26	0.00001	0.000010

Table 5.2: Actual and required area for standard binomial completion profiles

5.4 Analysis of the Generic Model

A detailed analysis of the behaviour of the generic model is presented in appendix B. This comprises simulation of the standard systems, described in section 5.3, with a range of parameter settings. This provides a representative sample of the behaviour of the model: three different systems are studied, each with a range of values for the probability of occurrence of detectable faults, and the occurrence of, and recovery from, hidden faults.

The effects of parameter changes on the behaviour of the generic model are summarised

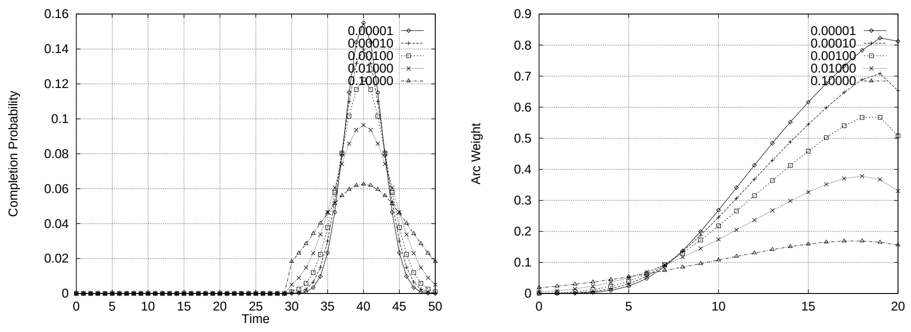

Figure 5.11: Standard binomial completion profiles, and corresponding arc weights

below, more detailed analysis is presented in appendix B.

- The *completed* state shows the following characteristics:
  - Increased p_d leads to a reduced probability density in the completed state.
  - Increased p_f leads to a reduced probability density in the completed state.
  - Increased p_r leads to a greater probability density in the completed state.

  These results are logical, and expected.

- The behaviour of the *hidden fault* is the inverse of that of the *completed* state. This is logical, since the two states model similar external events, differentiated only by the hidden internal state of the system.

- The behaviour of the *detectable fault* state shows the following characteristics:
  - Increased p_d leads to a reduced probability density in the detectable fault state.
  - Increased p_f leads to a reduced probability density in the detectable fault state.
  - Increased p_r leads to a greater probability density in the detectable fault state.

  The peak in the probability density plot towards the end of the execution of the system is noted: this is due to time overrun, with the systems which have not completed by this point being classified at failures. These results are logical, and expected.

- In a similar manner to the hidden fault and completed states, the *failed* state has behaviour which is the inverse of the detectable fault state.

Increasing the probability of detectable fault, $\mathrm{p_d}$, causes an increase in the probability distribution of the system into the *detectable fault* and *failed* states. This follows an exponential decay shape, due to the uniform failure probability, although this may be modified by the effects of the other transition probabilities.

Increasing the probability of hidden fault occurrence, $\mathrm{p_f}$, causes a switch from the *completed* and *detectable fault* states to the *hidden fault* and *failed* states. Once again, the exponential curve characteristic of a uniform probability is visible, although somewhat modulated by the completion profile of the system.

Increasing the probability of recovery from hidden faults, $\mathrm{p_r}$, reverses the effects of increasing $\mathrm{p_f}$.

The generic real-time system model is therefore seem to behave in a predictable manner; with combinations of parameters giving rise to logical changes in the system probability distributions.

5.5 Summary

In section 5.1 a novel model for the behaviour of real-time systems has been derived, based upon a number of observations of the behaviour of such systems, namely:

- Faults in a system may be classified as detectable or hidden: detectable faults lead to an observed system error, hidden faults transition the system to an alternate mode of operation, but cannot be immediately observed. Hidden faults must be detected by examination of the final output of the system, by means of some form of acceptance test. These properties lead to the derivation of a system model with two parallel sets of states, representing the two operational modes of the system.
- There is a limit to the accuracy of timing information which can be gathered about a real-time system: the resolution of typical clock devices is such that measurements of the execution-time of a system are discretised leaving a relatively small range of values. Hence, a discrete state model may be applied, with the progress of a computation being modelled as a series of probabilistic transitions along a state chain.
- Real-time systems have fixed execution time bounds, and exceeding those time bounds is an error. These time bounds limit the size of the model.
- Failures may occur at any time. Hence, transitions to the detectable fault states, and transitions between operational modes of the system, may occur at any time.

- Similarly, the system may complete its execution at a range of times. This is modelled by a set of probabilistic transitions to the completion states of the model.

The model resulting from these observations, figure 5.3, is both simple, and generic. The operation of a system is modelled as a series of transitions between the states of the model, from a single initial state to one of four final states, representing differing combinations of successful completion, completion with a hidden fault, and detectable failure of the system. The observed distribution between these final states illustrates the functional correctness of the system's operation. The temporal properties of the system are illustrated by the variation in this distribution with respect to time.

Estimation of the parameters of the model has been discussed in section 5.2, and it has been shown that these parameters are observable from system test data, allowing the behaviour of real systems to be modelled.

A number of testbed systems have been defined and their behaviour simulated. It has been shown that the behaviour of the model is both consistent and logical, as the model parameters are varied.

It is therefore seen that the model developed in this chapter allows for both the functional and temporal behaviour of a real-time system to be represented in a single high-level model. This has been derived from generic properties of real-time systems, and is hence independent of any specific design/implementation technique for such systems. This model is an improvement on traditional system reliability models which focus on functional correctness of a system and do not adequately model the temporal properties of that system.

The scope of application of this technique has deliberately been left ambiguous so far: this is discussed in chapter 6, where the application of this technique to systems modelling is discussed.

Chapter 6

Application to Systems Modelling

The work presented in chapter 5 develops a high level model of the properties of a generic real-time system. For such a model to be useful, it must be shown to be applicable in a predictive role: in addition to describing the behaviour of existing systems, the model must be able to predict the behaviour of new systems. Only a predictive model will be useful in the design and implementation of a fault tolerant embedded system. In this chapter the application of the generic model to predict the properties of two common fault tolerant techniques will be discussed. These techniques are the recovery block, and N-version programming. It is intended that the general applicability of this technique will be illustrated through these specific examples.

This chapter is divided as follows: section 6.1 discusses the derivation of a system model by means of combination of sub-system models. In sections 6.2 and 6.3 this technique is applied to the modelling of recovery block systems, and section 6.4 compares this to a number of other published recovery block models. Section 6.5 briefly discusses N-version programming systems, to show the scope of applicability of this technique. Finally, section 6.6 summarises and discusses the results presented in this chapter.

An analysis of a number of fault tolerant systems, such as that presented in this chapter, necessarily depends upon a number of studies of the reliability and failure probability information for such systems. Because of the volume of this material, comprising many system completion and failure probability graphs, it is felt that its inclusion in this chapter would be disruptive. For this reason, therefore, details of a number of the analyses undertaken in this chapter are provided in the appendices.

6.1 Derivation of System Models

The model described in chapter 5 provides an abstract description of a generic real-time system, and is derived from the fundamental properties of such systems. This model may be applied to predict the properties of actual systems, provided those systems behave in a manner compatible with the underlying assumptions of the model.

Most embedded control systems are complex entities, and this generic system model is not suitable for modelling such systems directly: application of the model to these systems would be a significant oversimplification. That does not mean, however, that the generic system model is of no use for predicting the behaviour of such systems, instead it must be employed as part of the modelling process.

As was discussed in chapter 2, techniques for achieving fault tolerance typically employ a number of diverse alternates. The results produced by these alternates are combined using some form of voting or acceptance test to produce a final output. The combination of multiple results leads to resilience in the presence of faults. It is these alternates to which the generic system model may be applied. They are relatively simple processes, typically developed and tested independently and later combined to form a fault tolerant system. Due to the independent testing process, and small scale of the alternates, it is a relatively simple matter to derive the parameters of the generic model for them.

It is then possible to model the complete fault tolerant system as a combination of these alternates. Each alternate is viewed as an instance of the generic system model, and these are combined to produce a model for the complete system. The remainder of this chapter illustrates this process for a number of different fault tolerant system architectures.

6.2 Recovery Block Model: Infallible Acceptance Test

The recovery block was described in section 2.3.2. It is a technique which uses multiple versions of a program block to attempt to ensure success in the presence of system failures. The first version is known as the *primary* and the second and subsequent versions are known as *alternates*. The primary is executed, and an acceptance test evaluated. If this fails, the alternates are executed in series until one succeeds. In order for the entire system to operate successfully under real-time constraints, it is necessary for each alternate to operate under such constraints. Each alternate in the recovery block may, therefore, be viewed as a generic real-time system, and the model developed in chapter 5 is applicable for modelling the behaviour of the primary and alternates.

In order to model the full recovery block, an acceptance test model is required in addition to the alternate model. This must map from the four output states of the generic model representing each alternate to the final pass/fail states of that alternate. A generic acceptance test will be fallible, that is, it will not correctly classify all systems, and will take a finite amount of time. Systems with such fallible acceptance tests are discussed in section 6.3. For reasons of simplicity and tractability of the analysis, however, the test modelled here will, initially, be assumed to be infallible and to take unit time.

The purpose of an acceptance test is to determine the correctness, or otherwise, of the result produced by an alternate in the recovery block. The generic system model has four final states, indicating varying degrees of correctness of the system output. An acceptance test model must map from these four states, to either an alternate *pass* state or an alternate *fail* state. If the acceptance test is assumed infallible, it will simply map from the *completed* state of the generic model to the alternate *pass* state. The three other final states of the generic model will be mapped onto the alternate *fail* state. If it is further assumed that the acceptance test will take unit time, only two additional states must be added to the generic model, together with single transitions from the final states of the generic system model to these new states. This combined alternate and acceptance test model is illustrated in figure 6.1.

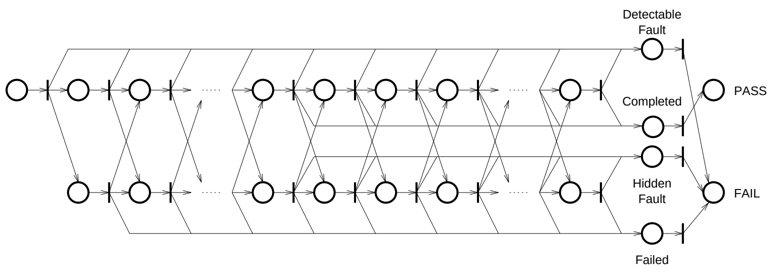

Figure 6.1: Alternate Model With Infallible Acceptance Test

It is clear that the behaviour of an alternate and acceptance test model such as this will be similar to that of the generic system model described in chapter 5. The probability profile observed at the pass state of this model will match that of observed at the completed state of the generic model, figure 5.3. In addition, the probability profile at the failure state of this model will be the sum of the three failure states of the generic system model.

By combining several such alternate/acceptance test models, a complete model of the recovery block may be derived, an example of this is shown in figure 6.2 for a recovery block consisting of a primary module, and two alternates. The combination of several alternate models into a model for the complete recovery block proceeds as follows:

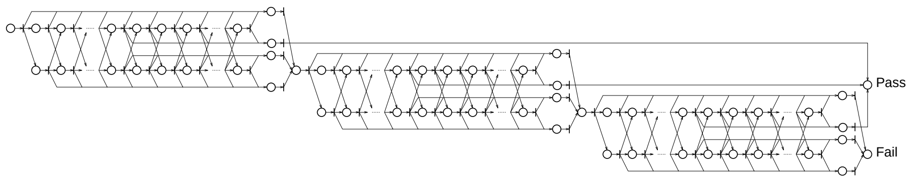

Figure 6.2: Recovery Block Model

each alternate has an acceptance test, and if any one of these tests succeeds the entire recovery block succeeds. Therefore, entry into any one of the alternate *pass* states is equivalent, and hence these states are merged into a single recovery block pass state, labelled *pass* in figure 6.2. Entry into an alternate *fail* state leads to the execution of the next alternate in series, hence the fail state of alternate n is merged with the initial state of alternate $n+1$. The *fail* state of the final alternate becomes the failure state of the entire recovery block. The result of this process is a simple, hierarchical, model for a recovery block system, which directly models the hierarchical structure of the software.

6.2.1 Sample Analysis

In order to show the possibilities inherent in this recovery block model, a sample system has been defined, and this system is subjected to analysis. This sample recovery block system comprises a primary module, and two alternates[1], and therefore corresponds to the model illustrated in figure 6.2.

This sample system is not based around any specific system. Rather, it comprises three alternates, each of which has properties such as those which *could* be found in actual real-time systems. These alternates are as follows:

- *Primary*: A slow but reliable system, where the completion probability increases with time. For example, some form of iterative solution or stepwise refinement technique.
- *1st Alternate*: A fast but unreliable system. For example a naive linear interpolation algorithm applied to a nonlinear system.
- *2nd Alternate*: A reliable, medium speed system. The completion profile of this system follows a "bell-shaped" curve. For example an algebraic solution to a set of equations, where the completion time is somewhat data dependent.

[1]It is noted that these techniques are not limited to recovery blocks with three alternates, this is merely a convenient size system for the purposes of example.

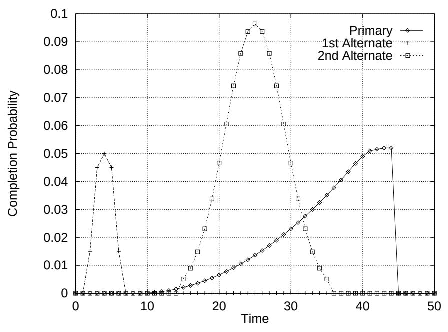

Figure 6.3: Basic Alternate Completion Profiles

	p_f	p_r
Primary	0.001	0.001
1st Alternate	0.010	0.010
2nd Alternate	0.010	0.005

Table 6.1: Alternate Parameters

It is noted, once again, that these alternates have been chosen for their illustrative value only, and are intended to illustrate the potential of a generic technique, rather than model the specifics of a single system.

The completion profiles chosen for these alternates are illustrated in figure 6.3. By application of the techniques discussed in section 5.2, in particular equation 5.5, these completion profiles may be converted to the necessary, p_c, arc weights for the alternate models.

In addition to these arc weights, the alternate model requires three other parameters to be defined: the probability of detectable fault, p_d; the probability of hidden fault, p_f; and the probability of hidden recovery, p_r. For the purpose of these tests the parameters p_f and p_r were fixed for each alternate, and p_d was varied. This corresponds, for example, to a system subject to variable rate transient hardware failure, where such failures cause detectable faults; and also subject to constant rate software failure/recovery processes. The values chosen for for parameters p_f and p_r are shown in table 6.1. These values are chosen to allow numeric results to be obtained: they are not necessarily representative[2].

[2]As was discussed in section 5.4, the generic real-time system model, upon which this recovery block model is based, shows linear changes in its behaviour as parameter values are modified. This implies that although the precise values obtained from these sample systems may not be representative due to the arbitrary choice of parameters, these values will scale to match real parameter values with no major behaviour changes.

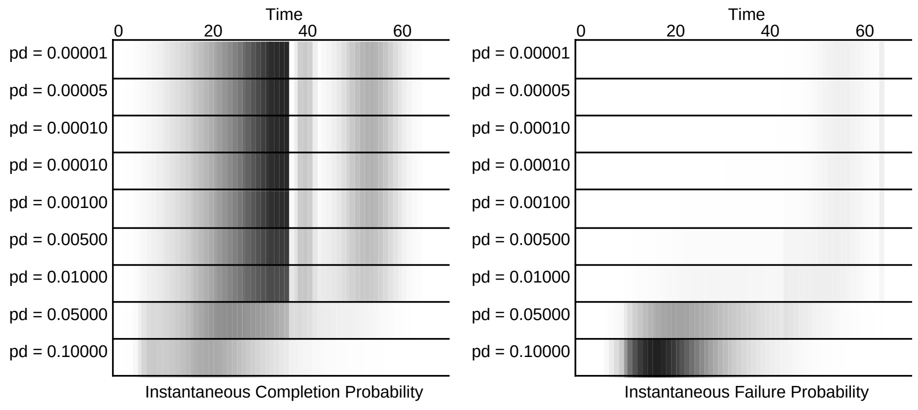

Figure 6.4: Recovery Block Completion/Failure Probability *vs.* Time, as p_d is varied

The results of this analysis comprise the instantaneous completion and failure probability, with respect to time, for the recovery block system. These are shown graphically in figure 6.4 for different values of the forward failure rate, p_d. The format of these figures is as follows: each row of the figure comprises the instantaneous probability for a particular value of p_d. Within the row, a darker shade of grey indicates greater probability. Values are comparable between rows, even though the precise numeric value of each point is not noted on the figure. The essential point of these figures is to show points during the execution of a system at which it is most likely to complete/fail.

The effects of an increased forward failure rate, p_d, are clearly shown in the completion probability curve, figure 6.4 left: when the forward failure rate, p_d, is small, the alternates are executing almost sequentially, with only the time overrun causing failures. Three regions of high completion probability are visible in the figure, corresponding to the peaks in the completion profile of the individual alternates in the system. As the failure rate increases, the shape of the recovery block's completion profile also changes: those systems which complete successfully do so sooner, but the completion probability decreases also. Further, the three alternates become less distinguishable.

The graph of recovery block failure probability with respect to time, figure 6.4 right, shows a related trend. At low failure rates, most failures occur towards the end of the system's life, due to exhaustion of alternates in the recovery block. As the failure rate increases, it is seen that the overall completion probability decreases, as is expected. In addition, the instantaneous failure profile no longer follows the pattern of sequential execution of the alternates: it is clear that alternates are failing early, and that the following alternates are being executed with various start times. The result is a completion profile which observes a more continuous distribution, rather than the three discrete peaks due to the individual alternates, and a failure profile which shows a broad spread of times when system failure is likely.

Comparing the completion and failure probability curves, it may be seen that the recovery block system exhibits two operational modes. At low forward failure rate, the system's behaviour is determined almost exclusively by time-overrun of alternates, and eventual exhaustion of alternates towards the end of the system's life. As the failure rate increases, there is a transition to a mode where alternates rarely complete their execution, and the system failure rate increases dramatically. It is clear that the designer of a fault-tolerant system comprising a number of recovery blocks must be aware of this phenomenon, and must take steps to ensure that the operational mode of such a system is well understood, since the scheduling of such a system will be affected to a large degree by this.

6.2.2 Recovery Block Timing Properties

A recovery block should be designed to maximise the probability that a system completes its operation successfully and before its deadline. In addition, it should ensure that system failures are noticeable as early as possible, to ensure that there is time for higher level recovery to take place. These two goals are not necessarily compatible: for a specific set of alternates, the order in which those alternates are combined to form a recovery block will affect the completion/failure time distribution. The effect of changing the order of execution of the alternates within the recovery block may be modelled using the technique described herein, and hence the most appropriate ordering of alternates may be chosen for a particular application.

It is noted that this is currently feasible for small numbers of alternates only, since the number of possible orderings for n alternates increases with $n!$ making simulations computationally infeasible for all apart from small n. In practice this is not a major limitation, since the majority of n-version systems deployed have a relatively small number of versions. In addition, it is unclear that large numbers of alternates increase a system's reliability in the general case. For example, the work of Eckhardt & Lee [25] illustrates a number of possible systems where increased numbers of alternates actually leads to *reduced* system reliability. Further, in many cases there exists an asymptotic limit to system reliability which is approached as the number of alternates is increased; this is illustrated in the work of Tomek *et al.* [90].

In order to illustrate this technique, the sample recovery block system used in section 6.2.1 has, once again, been used as a testbed. Since this comprises three alternates, there are a total of six possible orderings in which these alternates can be executed, as is illustrated in table 6.2. In this table, the primary is numbered 1, and the two alternates

	1st	2nd	3rd
RB1	1	2	3
RB2	1	3	2
RB3	2	1	3
RB4	2	3	1
RB5	3	2	1
RB6	3	1	2

Table 6.2: Possible Orderings for a Three Alternate Recovery Block

are numbered 2 and 3 respectively. The resulting six orderings provide six recovery block systems, RB1 to RB6.

For each of these systems, the behaviour of the recovery block system has been simulated for several values of p_d. The completion profiles of these systems are shown in figure 6.5. As can be seen, there is a large variation in the behaviour of the system dependent on the order in which the alternates are executed, and the forward failure probability, p_d. Since each alternate has a unique completion profile, the recovery block completion profile changes, depending on the order of execution of the alternates.

If the cumulative completion profiles for different recovery block systems are compared, the effects of different alternate orderings become clearer. Whilst the overall completion probability of the system is unaffected, it is clear that the likely completion time of the system will vary significantly depending on the chosen order of execution of the alternates. This is illustrated in figure 6.6, where it is seen that the cumulative completion probability increases at different rates for the different alternate orderings.

The effects of increasing the forward failure rate, p_d, are twofold, and may be observed in figure 6.7. Firstly, it is clear the increasing p_d leads to a reduced overall system completion probability. Secondly, it is seen that increased p_d leads to a greater divergence between the different alternate orderings. That is, a greater failure rate makes the ordering of the alternates more important for determining the system's completion time.

The failure probability information for the recovery block systems is shown in figure 6.8, with the instantaneous failure probability on the left, and cumulative on the right. The most interesting feature of this is that for a given failure rate the system failure profile is fixed, independent of the order of execution of the alternates (and hence only the RB1 system is shown, to conserve space). At first, this appears surprising, since the completion profile of the system changes with the differing orderings of alternates, the failure profile might reasonably be expected to do so too.

In practice this is not so, due to a fundamental assumption of the recovery block model:

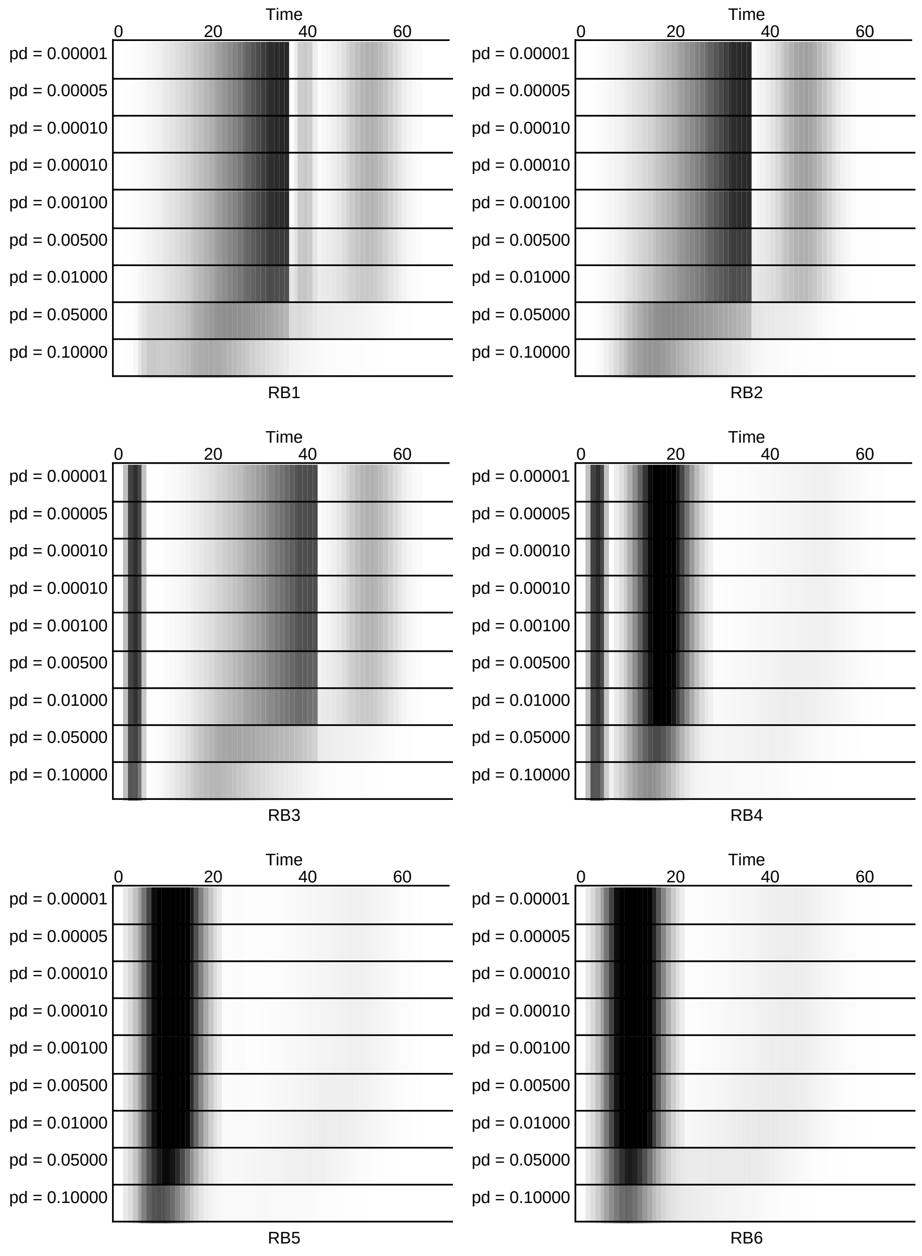

Figure 6.5: Recovery Block Instantaneous Completion Probability for different p_d

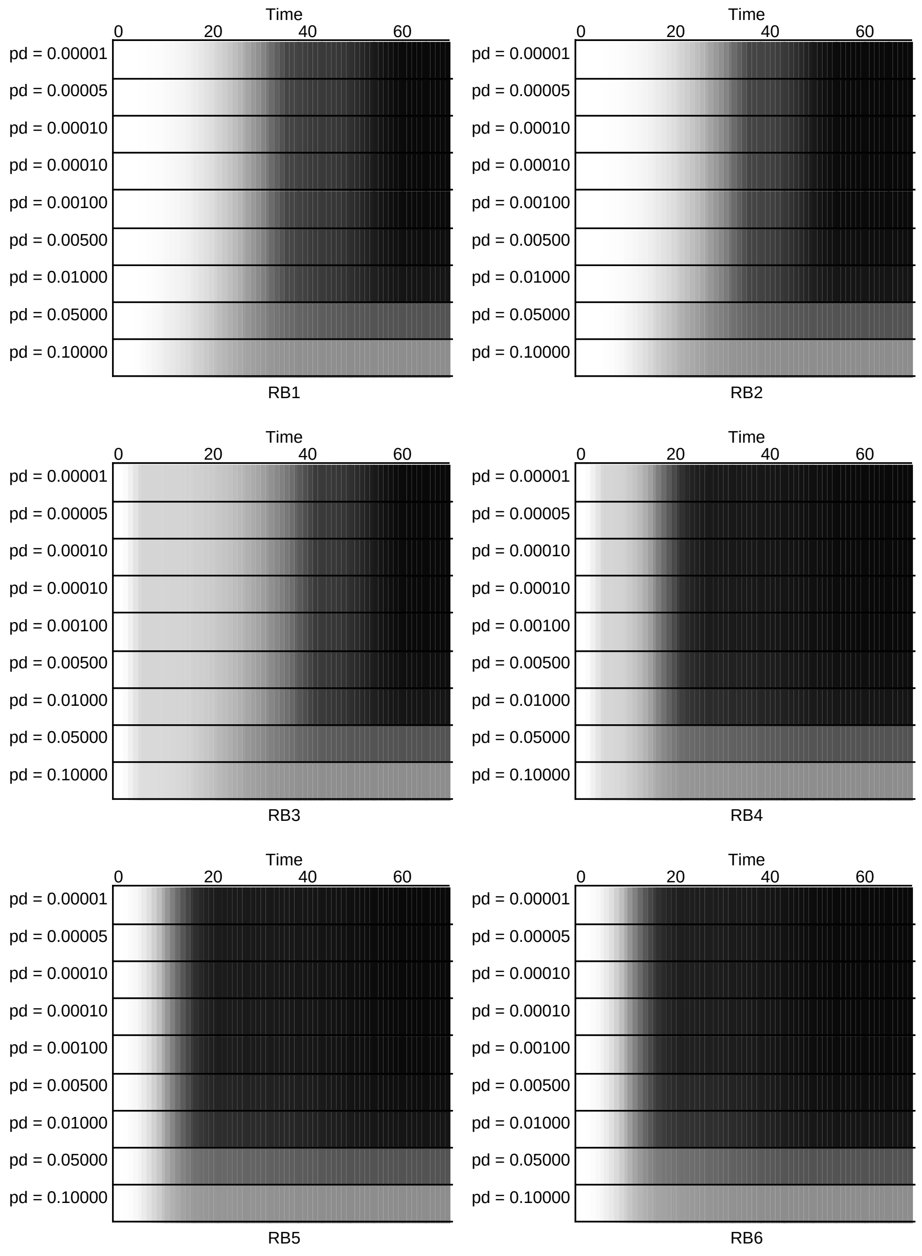

Figure 6.6: Recovery Block Cumulative Completion Probability for different p_d

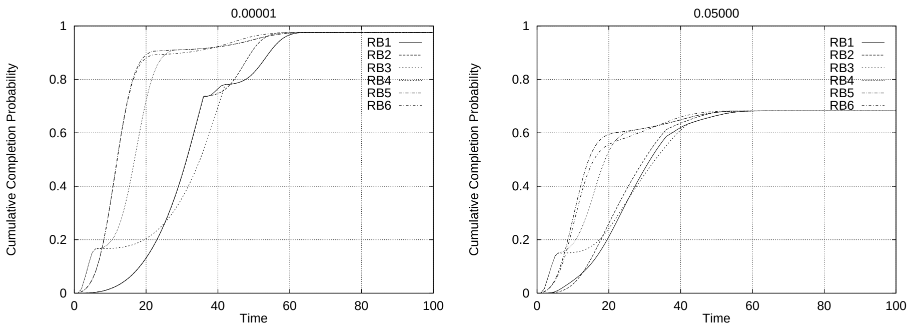

Figure 6.7: Effects of p_d on cumulative completion probability

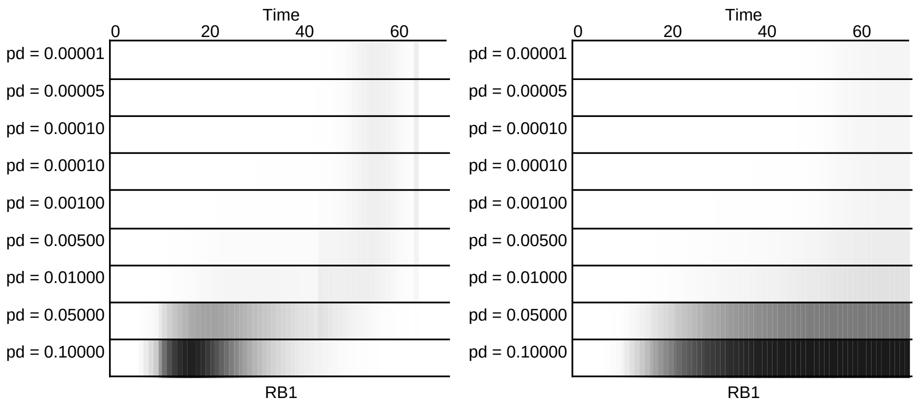

Figure 6.8: Recovery Block Failure Probability

independence of the alternates. The recovery block model is comprised of a number of alternates, each of which is modelled using a technique based on the generic system model described in chapter 5. As was discussed in section 6.1, the parameters for these alternates are derived in isolation, and then combined to form the complete recovery block model. These parameters hence do not include any effects due to interaction between the alternates when they are combined into the recovery block: it is assumed that is the performance of an alternate measured in isolation is the same as its performance when used in the recovery block.

Each alternate may be viewed as a function transforming an input stream. This input stream comprises the set of data points input to the system, and the alternate transforms this to produce two probability distributions at its output states. The alternates perform this function, irrespective of the source of their input data. Execution of the alternates is commutative, hence the probability distribution observed at the failure state of the recovery block, which has been processed by all alternates, is independent of the order of those alternates. On the other hand, the distribution at the completed state of the recovery block is only processes by a subset of the alternates, and hence varies with the order of execution of the alternates.

Given this assumption of independence, these results seem justified. It is, however, an open question whether this assumption holds; this is discussed in section 6.2.3.

6.2.2.1 Mean Completion Time

It is possible to derive a number of metrics from completion/failure profiles such as those illustrated in figures 6.5 to 6.8. The most obvious of these is the overall system completion probability, defined as the final value of the cumulative completion probability. The overall system failure probability is similarly defined. It is also possible to derive the mean completion/failure time; this is now discussed in some detail.

The mean completion time is here defined as the time at which the cumulative completion probability equals 0.5. Since the recovery block model used is a discrete time model, linear interpolation is used if the mean completion time falls between two time steps. This metric has been derived for the recovery block systems studied in section 6.2.2, these results are shown in figure 6.9.

As can be seen, for systems where the forward failure probability, $p_d < 0.01$, the mean completion time is effectively constant. However, as the forward failure probability increases above this threshold the mean completion time begins to decrease noticeably.

The six recovery block systems comprise three pairs: RB1 and RB2 with alternate 1 executing first, RB3 and RB4 with alternate 2 executing first, and RB5 and RB6 with

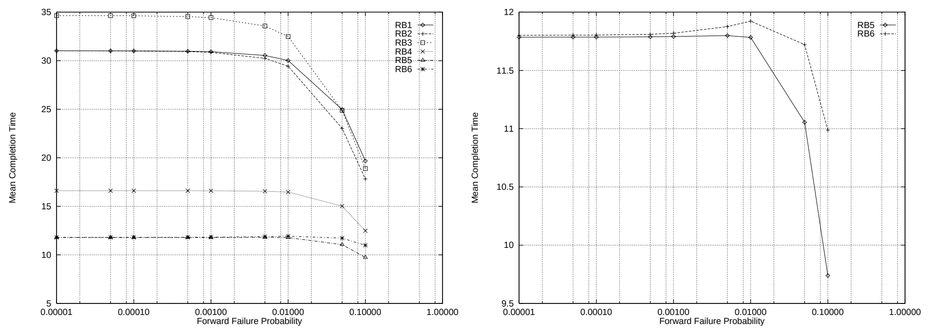

Figure 6.9: Mean Completion Time

alternate 3 executing first. It is noted that the RB3 and RB4 systems show a proportionally greater rate of decline than the others. These are the systems where alternate 2 is executed first, this being the least reliable of the three alternates. In the RB3 system the least reliable alternate is executed first, and the others follow in order of increasing reliability. This results in a system which has initially the largest mean completion time, but which exhibits the greatest rate of decline. The RB4 system has, once again, the least reliable alternate executing first, but the order of the other two alternates is reversed. This results in a system which has an initially lower mean completion time, and which exhibits a smaller rate of decline as the forward failure probability is increased. This is logical, with an initially unreliable alternate, increased failure probability results in a system which fails sooner.

The behaviour of the RB5 and RB6 systems is unusual: the mean completion time increases as the failure rate is increased. This is the counterpoint to the RB3 and RB4 systems: here the most reliable alternate is executed first. In the RB6 system, the alternates are executing in decreasing order of reliability, and this results in a system which shows not only the smallest rate of decline of the mean completion time but, in fact, a noticeable peak with increased mean completion time. The initial mean completion time is due mainly to time overrun of the first alternate, as the failure rate is increased a combination of the execution time of all of the alternates results in an increase over that of the first alternate only.

It is desired that a recovery block system either produce a correct result, or fail as soon as possible, in order that there is sufficient time for a higher level recovery procedure to operate before the system's deadline is exceeded. In order to achieve this for the systems studied, the alternates should be executed with a reliable alternate first, followed by the least reliable, and the others in increasing order of reliability. In this case the RB2 and RB5 systems exhibit this behaviour.

6.2.2.2 Mean Time to Failure

In a similar manner, it is possible to plot the mean time to failure for the recovery block systems studied in section 6.2.2: this is illustrated in figure 6.10. These results are unsurprising, as the forward failure probability is increased the system fails sooner. With reference to the failure profiles of these systems, figure 6.8, it is clear that the decline in the mean time to failure occurs at the point where the system changes its mode of operation away from being limited by time overrun.

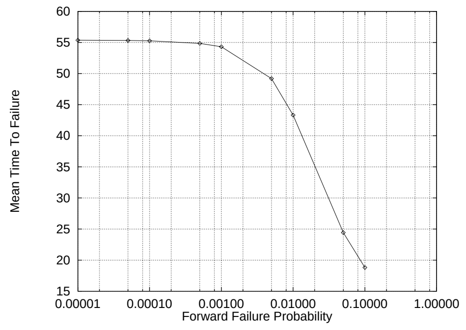

Figure 6.10: Mean Time To Failure

6.2.2.3 Recovery Block Timing Properties: Summary

A further discussion of the timing properties of the recovery block model may be found in appendix C. This provides further data to validate the conclusions drawn in this section, by means of a study of the behaviour of two additional recovery block systems.

It is clear that the order of execution of the alternates has a profound effect on the timing properties of a recovery block system. In cases where the probability of detectable faults occurring is low, it is seen that the alternates execute in an essentially sequential fashion: virtually all failures are due to the time overrun of the alternates, and the instantaneous completion probability plots show that the completion profile for the complete recovery block system is comprised of those of each alternate concatenated in a sequential fashion.

As the probability of detectable faults occurring increases, the alternates no longer execute in such a clearly defined fashion: more failures occur, and the fixed time bounds between execution of the different alternates are blurred. In this mode of operation both completion and failure of the recovery block are likely to occur sooner, and the failure probability increases greatly.

It is seen that the failure time is independent of the order of execution of the alternates of the recovery block system; but the completion time is not. Therefore, care should be taken to ensure that the alternates are executed in an order which encourages early completion. Clearly this order will depend on the precise nature of the alternates used, and use of the techniques discussed herein can enable determination of the optimum order based on test data for the individual alternates.

It is possible to determine a number of guidelines for the best ordering of the alternates for a particular recovery block. In order to ensure that the sample system is likely to complete early, the *most* reliable alternate should be executed first, followed by the *least* reliable, and then the others. Similar results can be derived for other recovery block systems, and further examples are presented in appendix C.

6.2.3 The Effects of Dependent Alternate Failure Rates

So far, it has been assumed that the alternates in a recovery block are independent: that is, the failure rate of an alternate measured in isolation is the same as its failure rate when employed in the recovery block. It has, however, been shown that this assumption is likely to be invalid. For example, the work of Eckhardt & Lee on N-version programming systems [25] states that

> "...recent experiments have demonstrated that programmers given the same task are prone to make mistakes that potentially reduce the effectiveness of a fault-tolerant approach. These mistakes, although perhaps totally unrelated, appear in the application environment as coincident failures; that is, two or more versions fail when operating on the same input."

Further, it is shown that the failure intensities required of the components in a system, for the assumption of independence to hold, are unlikely to be found in practice. An extension of Eckhardt & Lee's work by Tomek [90] has concluded that

> "...even independently developed modules are prone to exhibit the same types of errors when operating on the same input."

In addition the oft-cited work of Knight and Leveson [46] indicates that the alternates in an N-version programming system (section 2.2.1) are likely to fail in a coincident manner.

The recovery block has a number of features in common with N-version programming systems, and it is expected that the concerns expressed in the papers cited above are equally valid for recovery block systems, since both utilise redundant multiversion software. There is, however, an important difference: in an N-version programming system, all versions of the software receive the same set of inputs, and execute in parallel. In a recovery block system, the primary receives the full set of inputs, whilst the secondary only processes those points for which the primary failed. Similarly, the third alternate receives only those inputs for which the primary and secondary have failed, etc. This effect has been studied theoretically by Csenki [21], who derives a model for the expected number of input points processed by a recovery block before failure: the failure probabilities for each alternate are parameters to this model, but the likely values of these parameters are, unfortunately, not discussed. The effects of these interactions between the alternates in a recovery block are further studied by Laprie and Kanoun [50] who state that

> "a distinction has to be made between the interface failures (characterising the failures occurring during interactions with other components) and the internal component failure rates."

This work further suggests that the interface failure probabilities are at least as important as the internal component failures rates in determining the failure rate for a component.

It is clear, therefore, that the effects of coincident, correlated, failures between the alternates in a recovery block must be studied.

The recovery block model developed in section 6.2 has four parameters which may be influenced by the effects of dependent alternates. These parameters are: the completion profile for each alternate, p_c, the probability of detectable fault, p_d, and the probabilities of hidden fault and recovery, p_f and p_r.

As was discussed in section 5.2, the completion profile of an alternate is defined independent of the occurrence of faults, and must be derived from knowledge of the algorithm used by the alternate and/or test data. This expresses the fundamental behaviour of the alternate, and hence is not affected by the effects of coincident faults.

The other three parameters are, however, candidates for modification due to these effects. These parameters fall into two categories: those representing detectable faults, p_d, and those representing hidden faults, p_f and p_r. In section 6.2.3.1, the effects of changes in p_d are studied, whilst section 6.2.3.2 discusses the effects of modifications to p_f and p_r.

6.2.3.1 The Effects of Dependent Alternates: p_d

The behaviour recovery block described in section 6.2.2 has been studied with the addition of coincident faults, dependent on the probability of detectable fault, p_d. Such faults are modelled using a failure probability multiplier, p_{d_m}, by which the probability of detectable fault, p_d, is multiplied in subsequent alternates. That is, the probability of detectable fault in alternate n, p_d^n, is defined as in equation 6.1:

$$p_d^n = p_{d_m} \cdot p_d^{n-1} \tag{6.1}$$

It is assumed that the effects of dependent faults are to increase the failure probability of an alternate, hence it is expected that p_{d_m} will be greater than unity. For the purpose of these tests the failure probability multiplier, p_{d_m} was varied such that $1.0 \leq p_{d_m} \leq 1.5$, giving up to a 50% increased forward failure rate for each additional alternate. The details of this analysis are presented in appendix D.1.

The effects of increasing p_{d_m} are clear and consistent: as p_{d_m} is increased, the completion probability is reduced, and the failure probability correspondingly increased. It is noted that the basic shape of the completion and failure probability curves is not affected. The amount by which the completion and failure probabilities vary is seen to be dependent on the value of p_d. For small p_d there is little effect, but as p_d is increased the probability plots show greater dispersion.

Changing the value of p_{d_m} does not affect the similarity of the failure probability plots: in each case the same failure probability profile is produced, regardless of the ordering of alternate execution.

6.2.3.2 The Effects of Dependent Alternates: p_f

A further analysis of the recovery block model described in section 6.2.2 has been performed. This analysis, described in appendix D.2, includes a model of the effects of dependent alternate failure, modelled as an increase in the probability of hidden fault occurrence, p_f. This increase in p_f is represented as a failure probability multiplier, p_{f_m}, which is a small factor by which p_f is multiplied in each alternate. That is, the value of p_f in alternate n is defined by equation 6.2:

$$p_f^n = p_{f_m} \cdot p_f^{n-1} \tag{6.2}$$

As discussed in section 6.2.3, it is assumed that the effects of dependent faults are to increase the failure probability of an alternate, hence it is expected that p_{f_m} will be greater than unity. For the purpose of these tests, p_{f_m} was varied such that $1.0 \leq$

$p_{f_m} \leq 1.5$, giving up to a 50% increased hidden fault occurrence rate for each additional alternate. The details of this analysis are presented in appendix D.2.

It is clear that the effect of dependent alternate failure, modelled by an increased hidden fault rate, is to cause reduced system completion probability. The basic shape of the plot of completion probability against time is not affected, the likelihood of completion is reduced though.

Once again, it is seen that changing the value of p_{f_m} does not affect the similarity of the failure probability plots: in each case the same failure probability profile is produced, regardless of the ordering of alternate execution.

6.2.3.3 The Effects of Dependent Alternates: Summary

In sections 6.2.3.1 and 6.2.3.2 results from the analysis of recovery block systems subject to correlated alternate failure have been discussed. It has been seen that such correlated failures cause simple increases in the failure rate of the complete recovery block, but do not affect the basic timing properties of the system.

In addition, although not described above, it has been shown that combinations of dependent failure, modelled as increases in both p_d and p_f, result in predictable, linear combinations of the responses seen with single correlated failures.

It is, therefore, seen that whilst correlated failures are important when considering a system's reliability, they are less important when the timing properties of the system must be studied. This is important, since it allows scheduling decisions to be made, with some degree of reliability, before system integration and test are undertaken, and the full degree of failure correlation is observed.

6.2.4 Discussion

It has been shown that a recovery block system has two modes of operation: when the probability of detectable fault occurrence is small, alternates which fail before their time limit are unusual. This results in the completion profile for the complete recovery block being a concatenation of the completion profiles for the individual alternates. The failure profile of such a system shows very few failures occurring early in the execution of the recovery block, but a peak where a number of failures occur toward the end of the recovery blocks execution, due to the exhaustion of alternates.

As the probability of detectable faults increases, the system exhibits a transition to a second mode of operation: the alternates no longer run to completion, but rather fail earlier. This results in a system where the failure profile shows a much enlarged peak early in the system's execution. In addition, the completion profile no longer resembles the concatenation of the completion profiles for the individual alternates: since the alternates fail at different times, there are a range of start times, and hence completion times, for the second and subsequent alternates. This results in a more continuous completion profile for the recovery block, with the effects of the individual alternates being less noticeable.

This separation of the behaviour of a recovery block into two modes is confirmed by the mean completion and failure time data. It is clear that recovery block systems exhibit little change in their mean completion and failure time, until the probability that a detectable fault occurs exceeds a certain threshold value, at which point these metrics decrease rapidly. The point at which this occurs corresponds to the mode change in the behaviour of the system.

The order in which the alternates of a recovery block are executed has a significant effect on the behaviour of that recovery block. The overall reliability of the recovery block is not affected by these changes in alternate ordering, but the timing properties of the system are affected significantly. When designing a system with strict timing constraints, the knowledge of these changes in the timing properties of a recovery block dependent on the order of alternates execution will be useful.

It is desired that a recovery block system either produces a correct result, or fails as soon as possible, in order that there is sufficient time for a higher level recovery procedure to operate before the system's deadline is exceeded. To achieve this for the systems studied, it appears that the alternates should be executed with the most reliable first, followed by the least reliable, and the others in increasing order of reliability. Application of the techniques presented here will allow for similar recommendations to be produced for other recovery block systems.

Given information such as this, and a knowledge of the expected use of a system, the designer of that system is in a position to make an informed decision on whether the absolute worst-case execution time must be used, or whether a reduced set of bounds can be chosen, with a specific risk that the system will fail to perform within these bounds. In many cases the absolute worst-case behaviour is sufficiently unlikely, and the failure probabilities of other parts of the system are sufficiently large, that the increased probability that the system will exceed its deadline is acceptable. It is, therefore, seen that application of this model will provide greater confidence that software can be designed to a specific, *tolerable*, level of risk.

6.3 Recovery Block Model: Fallible Acceptance Test

A recovery block system will utilise an acceptance test to determine the correctness, or otherwise, of the results produced by each alternate. In the terms of the model developed in this thesis, the acceptance test maps from the four final states of the generic system model discussed in chapter 5 to the final pass/fail states of the alternate. Recovery block systems utilising infallible acceptance tests were discussed in section 6.2, however the possibility of an acceptance test which contains faults cannot be discounted, it is these *fallible* acceptance tests which are studied in this section.

There are four possible outcomes of the execution of an acceptance test, each of which have an associated occurrence probability [77]:

1. A correct result is accepted, probability p_{ca}.
2. A correct result is rejected, probability $1 - p_{ca}$.
3. An incorrect is result rejected, probability p_{ir}.
4. An incorrect is result accepted, probability $1 - p_{ir}$.

The combined alternate and acceptance test model required is a simple extension to that used previously, figure 6.1, where a number of additional arcs are added to represent the extra failure modes of the system. This new alternate and acceptance test model is illustrated in figure 6.11, the extra arcs have probabilities as follows:

- The arc from the *completed* state to the *pass* state has weight p_{ca}.
- The arc from the *completed* state to the *fail* state has weight $1 - p_{ca}$.
- Other arcs leading to the *fail* state have weight p_{ir}.
- Other arcs leading to the *pass* state have weight $1 - p_{ir}$.

The *infallible* acceptance tests studied in section 6.2 correspond to systems with parameters $p_{ca} = p_{ir} = 1.0$.

Recent work into the effectiveness of acceptance tests in helicopter flight control systems [27], classifies the results of an acceptance test in a similar fashion: *false alarms*, *undetected faults*, *detected faults* and *stop faults*. The results obtained in that work show that false alarms, corresponding to an acceptance test which rejects correct results, comprise less than 2% of the total detected faults.

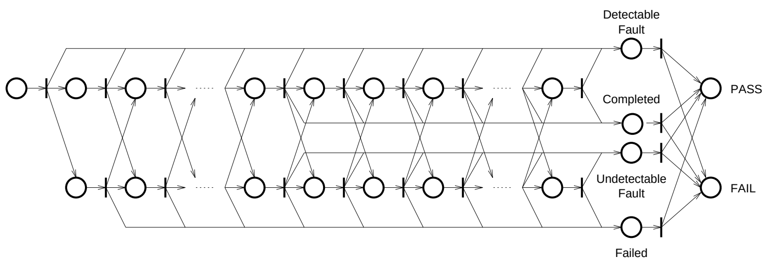

Figure 6.11: Alternate Model With Fallible Acceptance Test

The results vary somewhat, depending on the nature of the acceptance test used, but the sum of *stop faults* and *detected faults*, which corresponds to p_{ir} in the model developed in this thesis, is typically between 0.7 and 0.9. The *undetected fault* probability is around 0.1 to 0.3, corresponding to $1 - p_{ir}$ in this model. The *false alarm* rates are typically less than 0.02, corresponding to $1 - p_{ca}$. This leads to likely parameter values as shown in table 6.3.

Parameter	Likely range
p_{ca}	> 0.98
p_{ir}	$0.7 \cdots 0.9$

Table 6.3: Estimated acceptance test parameters

Simulation of the behaviour of the recovery block has been repeated, with $p_{ca} = 0.99$ and $p_{ir} = 0.8$, to match the values in table 6.3, and the results of this are shown in figures 6.12 to 6.14, the failure results are identical for all alternate orderings, so only the results for the RB1 system are shown.

The instantaneous completion profile, figure 6.12, shows predictable results: systems with infallible acceptance tests are more likely to be diagnosed as successfully completing their execution at any given time. The only exceptions to this are at the final time step of the execution of each individual alternate, when the system with fallible acceptance test shows greater completion probability. This is confirmed by the plots of cumulative completion probability, figure 6.13, which show greater cumulative completion probability for the systems with fallible acceptance test, after the completion of the first alternate. The failure probability results, figure 6.14, also show this trend.

These results appear strange at first: a fallible acceptance test results in a system which appears to be more reliable than that with a perfect acceptance test. A few moments thought should suffice to show that this result is, in fact, logical: with $p_{ca} = 0.99$ the vast majority of correct results are accepted as correct, however since $p_{ir} = 0.8$, some

20% of the incorrect results are also accepted as correct, resulting in a system which *appears* more reliable than that with a perfect acceptance test, due to misclassification of incorrect results as correct.

This is an important point, which should be noted by designers of safety critical systems: Due to imperfections in the acceptance test of a recovery block, or similar structure, a system may appear more reliable than is deserved. The possibility of misclassification of incorrect results as correct is very real, and it is clear that much work should be expended to ensure the reliability of the acceptance test.

6.4 Comparison with other Recovery Block Models

In this section, the model developed in sections 6.2 and 6.3 is compared with a number of other recovery block reliability models which have been developed in the literature. The aim is to show both the similarities between this model and other models, and also the differences. In addition, a number of models which have influenced the development of this model are discussed.

6.4.1 Eckhardt & Lee and Nicola & Goyal

The work of Eckhardt & Lee develops a theoretical basis for analysing the behaviour of multiversion software under the influence of coincident faults. This work is based around the definition on an *intensity function* which describes the probability that coincident faults are introduced into the diverse versions in an N-version programming system. The procedure developed in this paper for evaluating the failure probability for an N-version programming system is reliant on the nature of this intensity distribution: unfortunately, determining this distribution is a non-trivial problem.

There have been a number of attempts to find acceptable intensity distributions to enable the application of this work. The most noteworthy of these is the work of Nicola & Goyal [69] who use a beta-binomial intensity distribution to fit the software reliability data of Knight & Leveson [46, 47].

By application of a beta-binomial intensity distribution, Nicola and Goyal define the probability that i failures occur in an N version system as in equation 6.3:

$$b_N(i) = \binom{N}{i} \frac{(\pi + (i-1)\theta)(\pi + (1-2)\theta)\cdots\pi(\chi + (N-i-1)\theta)(\chi + (N-i-2)\theta)\cdots\chi}{(1 + (N-1)\theta)(1 + (N-2)\theta)\cdots 1} \tag{6.3}$$

The parameters to this model are as follows: π is the mean failure probability of a program version on a random input, $\chi = 1 - \pi$ is the mean completion probability of a

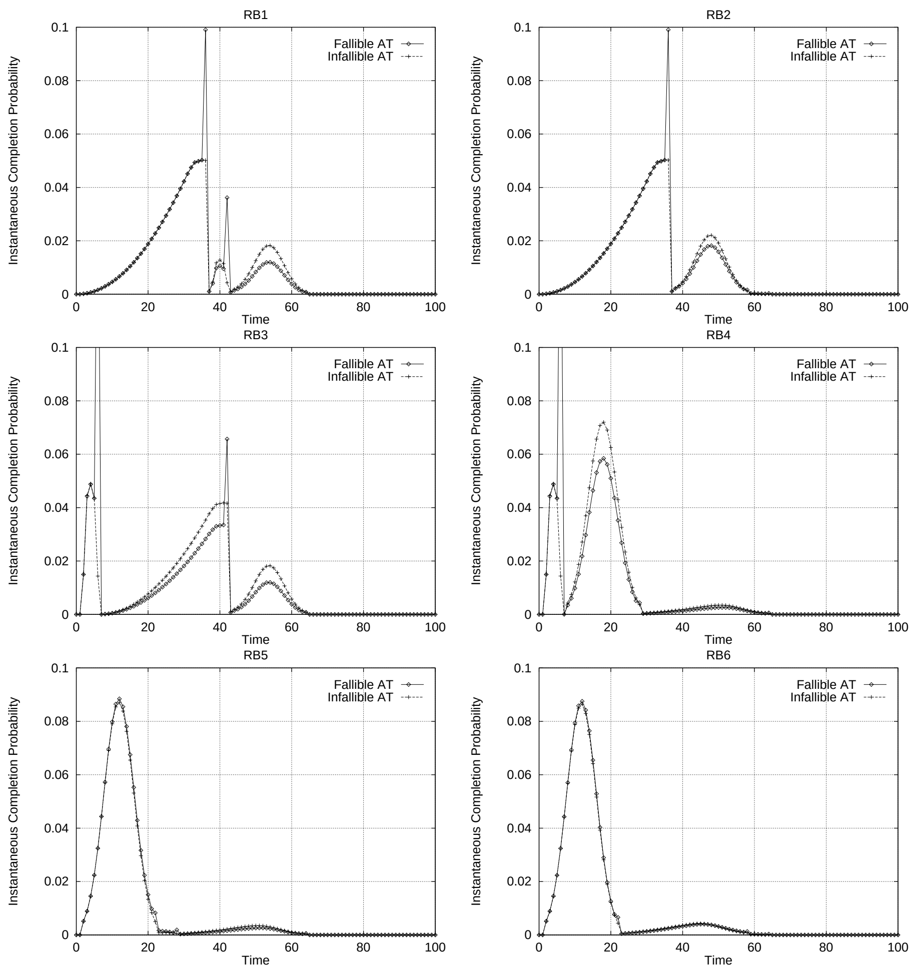

Figure 6.12: Effects of fallible acceptance tests: Instantaneous Completion Profile

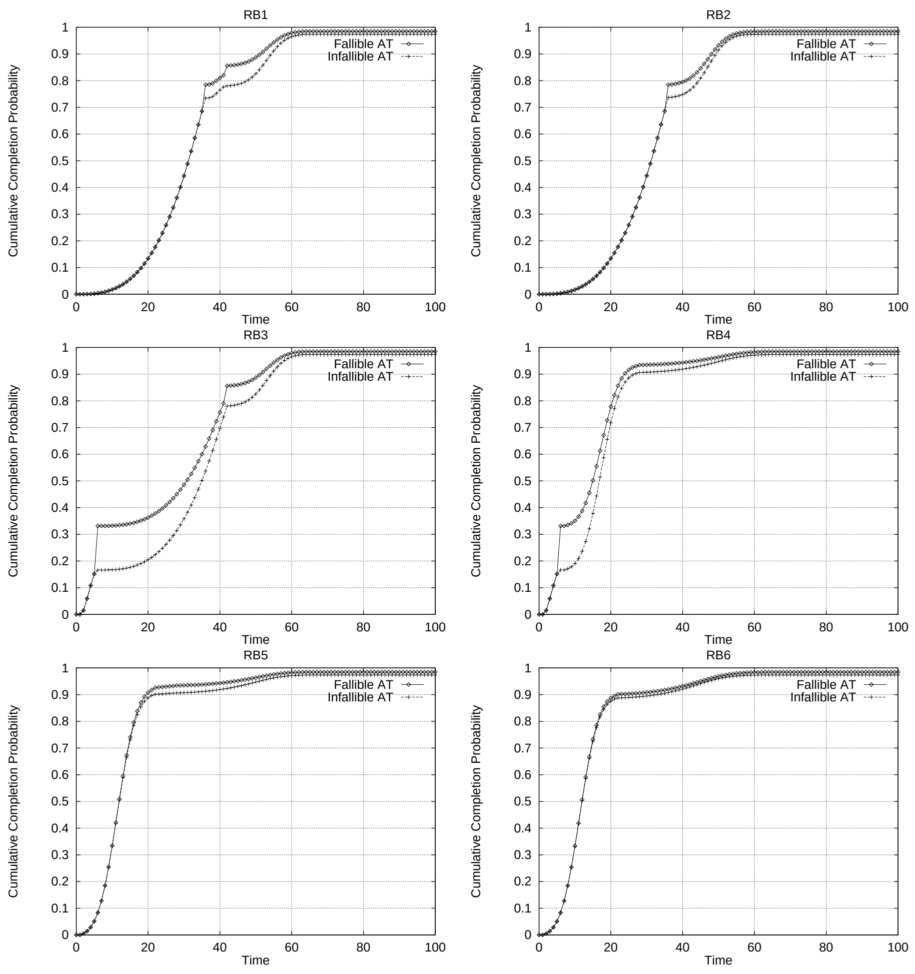

Figure 6.13: Effects of fallible acceptance tests: Cumulative Completion Profile

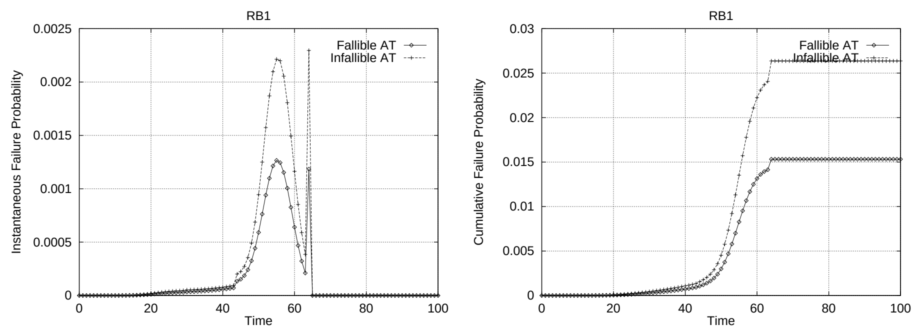

Figure 6.14: Effects of fallible acceptance tests: Failure Profiles

program version on random input, and $\theta : 0 \leq \theta \leq 1$ is the correlation among versions. It is noted that these parameters may be directly estimated from experimental data, indeed if the mean, μ, and variance, σ^2, of the number of alternates failing on a random input are known, then these parameters may be derived as in equations 6.4 and 6.5 [69]:

$$\mu = N\pi \tag{6.4}$$

$$\sigma^2 = N\pi\chi(1 + N\theta)/(1 + \theta) \tag{6.5}$$

In a recovery block system, failure of the recovery block only occurs when *all* of the alternates fail for a single input. In the model of Nicola & Goyal, this corresponds to the case when $i = N$. It is possible to model the behaviour of a recovery block comprised of three alternates using the Nicola & Goyal model. This is done by substituting $N = i = 3$ into equation 6.3, which then reduces to the form of equation 6.6:

$$b_{rb} = \frac{(\pi + 2\theta)(\pi + \theta)\pi}{(1 + 2\theta)(1 + \theta)} \tag{6.6}$$

This expression represents the failure probability for a three alternate recovery block system, parameterised by the mean failure probability of the alternates, π, and the degree of correlation between alternate failures, θ. A plot of the recovery block failure probability predicted by equation 6.6 is shown in figure 6.15 for all possible values of π and θ. The key at the top-right of this figure indicates the height of the contour lines plotted on the base; these indicate points of equal recovery block failure probability.

When applied to a three alternate recovery block system, the model of Nicola & Goyal depends upon two parameters only: the mean failure probability of a program version on random input, π, and the correlation amongst versions, θ. The effects of the correlated failure of alternates in a recovery block system have also been studied in this thesis, section 6.2.3, and the parameters of that model may easily be reduced to the form

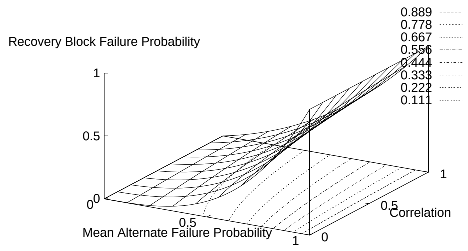

Figure 6.15: Recovery Block Failure Probability, based on Nicola & Goyal

required by the Nicola & Goyal model, allowing a comparison of the two models to be performed.

The alternates employed in section 6.2.3 to study the effects of correlated alternate failure in recovery blocks had failure probabilities as illustrated in table 6.4. These represent the probability that these alternates will fail when presented with a random input, with no correlated failures between the alternates. This information may be used to predict the failure probability of the entire recovery block, using the Nicola & Goyal model, if the mean value of the alternate reliability, the bottom row from table 6.4, is substituted for the parameter π in equation 6.6. This results in a prediction for the recovery block failure probabilities as illustrated in figure 6.16 (left), where the curves are labelled according to the base p_d value.

System	p_d	
	0.000500	0.050000
Primary	0.276713	0.788838
1st Alternate	0.833632	0.850036
2nd Alternate	0.115299	0.474073
Mean Value	0.408548	0.704316

Table 6.4: Alternate Failure Probabilities

By means of comparison, the failure probabilities for the systems studied in section 6.2.3.1 are illustrated in figure 6.16 (right) for two different values of p_d. The recovery block failure probability is here the final value of the cumulative failure probability for these systems, figures D.4 and D.8. The failure probability multiplier is the value of the parameter, p_{d_m}, and corresponds roughly to the correlation factor in the Nicola & Goyal model.

It is noted that there is no precise mapping between the correlation factor of Nicola &

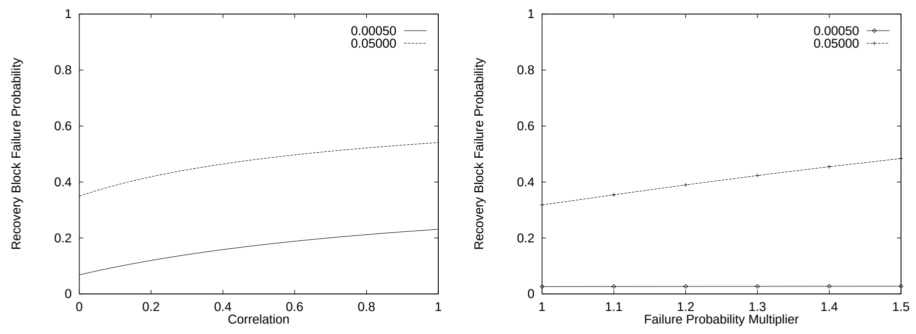

Figure 6.16: Comparison with Nicola & Goyal's model

Goyal, and the failure probability multiplier, p_{d_m}, used in section 6.2.3.1 of this thesis. In both cases, the initial value of the parameter indicates no correlation, and increased values indicate a greater degree of correlation between alternate failures. The correlation factor of Nicola & Goyal is expected to reside in the range $0 \ldots 1$; where as the failure probability multiplier, p_{d_m}, has a theoretically unlimited range, although in practice it is expected to be small. The results presented in figure 6.16 should not, therefore, be expected to show a precise match between the two models: similarities in the behaviour of the two systems are expected, although the details will differ due to the difference in parameter interpretation.

Despite the differences caused by the interpretation of the correlation parameters, it is clear that the systems where $p_d = 0.05000$ have similar behaviour predicted by both models. The systems with $p_d = 0.00050$, however, show quite distinct behaviour. The model of Nicola & Goyal uses the mean alternate failure probability, in contrast to the model developed in this thesis which treats each alternate separately. The alternates used in this test show a wide variation in their failure probability, this variation being greatest when p_d has small values. As is shown in table 6.4, the alternates failure probabilities for the case where $p_d = 0.00050$ are scattered towards the two extremes of reliability: this leads to a mean which is dissimilar to the failure probability of any one alternate. The large variation in the behaviour of the two models for this system is caused by this variation in the alternate failure probabilities, causing the Nicola & Goyal model to overestimate the recovery block failure probability.

The model of Nicola & Goyal is therefore seen to be in agreement with the model developed in this thesis, for systems where all alternates in the recovery block have similar failure probabilities. For recovery block systems where the reliability of some alternates is significantly different to the others, the two models disagree. This is due to the approximation of a number of system parameters by a single metric in the Nicola & Goyal model, and represents a deliberate simplification, and limitation, of their model. The advantage gained by the Nicola & Goyal model as a result of this, is that the parameters of the model are easier to estimate than those required by the model developed in this thesis.

6.4.2 Pucci

The model of Pucci [77] has a number of similarities to the recovery block model developed in this thesis, but also a significant set of differences. The essential similarities are the nature of the fault model employed, and the classification of faults. The differences are due to the timing properties of the models.

The model developed by Pucci describes a recovery block using a Markov chain, with the states of the chain corresponding to the execution of the alternates under different error conditions. This model has

- one state per alternate to represent the successful execution of that alternate
- one state for each alternate, except the first, to represent execution of that alternate when the preceeding alternate has failed
- three additional states to represent undetected failure (UF), acceptance test failure (ATF), and failure due to lack of alternates (AF)

An example of this model for a three alternate recovery block system is illustrated in figure 6.17:

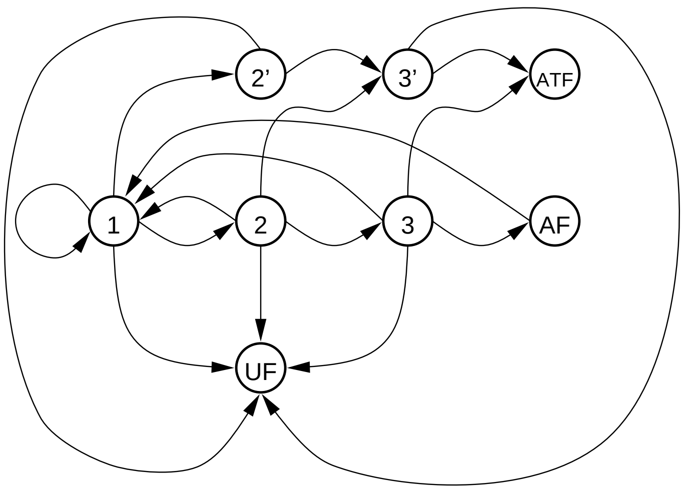

Figure 6.17: Pucci's Recovery Block Model

The arc weights for Pucci's model are assigned as follows:

- Transitions from state i to state 1 model the successful execution of the i-th alternate. That is, the alternate delivers correct results which are accepted by the acceptance test.
- Transitions from state i to state $i + 1$ represent the invocation of the $(i + 1)$-th alternate after the i-th alternate has failed. The alternate delivers incorrect results which are rejected by the acceptance test. If the last alternate fails in this manner, the system moves to the AF state, representing a *detectable failure* of the recovery block.
- A transition from state i to state UF represents a hidden fault in alternate i. The alternate has produced an incorrect result, which is accepted by the acceptance test. This is a *hidden failure* of the recovery block.
- A transition from state i to state $(i + 1)'$ models an acceptance test failure: a correct result is rejected by the acceptance test. The system continues with the execution of the next alternate. If there are no more alternates the system enters state ATF, representing a *correct* system which the acceptance test has classified as having a *detectable fault*.

It is clear that this model is similar to the model developed in this thesis. In particular, the concept of a parallel state chain representing the execution of a system in the presence of hidden faults, section 5.1, was a particular source of inspiration for the model developed here. The definition of the final states of Pucci's model also mirrors that of the model developed herein.

There are also a number of differences in the two models: in particular, the model of Pucci does not include any discussion of the timing properties of the alternates. In his paper Pucci derives an expression for the

> "...number of correct executions of the [recovery block] before any failure, or undetected failure..."

and for the mean time to failure for the recovery block. In each case, however, this is modelled only as the number of alternates executed, and discussion of the timing properties of the individual alternates are omitted. This is the major difference in scope of the two models: the model developed in this thesis allows for more precise determination of the timing properties of a recovery block system, the model of Pucci is simpler, but provides significantly less detailed information.

6.4.3 Csenki

The work of Csenki [21] discusses the reliability of recovery block systems where failure points in the input space are assumed to be clustered together. This work is notable since it provides a theoretical model for the filtering effect of the alternates in a recovery block, where alternate n processes only those points on which the preceeding $(n - 1)$ alternates have failed.

The model of Csenki is based around an infinite two-dimensional Markov chain, stepwise elimination is performed to reduce the state space of this chain, and expressions for a number of model parameters are derived: these parameters are the mean and variance of the number of successfully processed input points. It is noted that these expressions are themselves infinite sums, and an estimate of the truncation error is provided. The parameters required by Csenki's model are as follows:

1. The spontaneous failure probability of each alternate.
2. The number of consequential failures subsequent to a random failure of the primary.

The first set of parameters should be relatively simple to estimate, however estimation of the other parameters is a difficult problem. Indeed, Csenki states that

> "...no real data-set could be found matching the structure of [the model], nor was it feasible to set up experiments."

It is therefore seen that Csenki's model provides an important theoretical introduction to the concept of failure clustering, and the filtering effects of multiple alternates in recovery block systems. It does not, however, provide predictions amenable to analysis.

6.4.4 Laprie & Kanoun

The work of Laprie & Kanoun [50] is based on the assumption that

> "...classical reliability theory can be extended in order to be interpreted from both hardware and software viewpoints..."

and that

> "...even though the action mechanisms of the various classes of faults may be different from a physical viewpoint according to their causes, a single formulation can be used from the reliability modelling and statistical estimation viewpoints."

The essential thesis of [50] is that, so far as is required for reliability modelling, the cause of a fault is unimportant, provided a statistical estimate of the failure distribution can be obtained. The concept of a random fault model, as used in section 4.1 of this thesis, is studied, and in particular the use of a random fault model of software is suggested:

> "In the case of software, the randomness comes at least from the trajectory in the input space which will activate the faults. In addition, it is now known that most of the software faults which are still present in operation, after validation, are "soft" faults, in the sense that their activation conditions are extremely difficult to reproduce; hence the difficulty of diagnosing and removing them, which adds to the randomness."

A system comprising both hardware, subject to classical random failure modes, and software, is here modelled using a simple random fault model.

From this simple basis, a hierarchical system reliability model is derived, with the limit of the hierarchy being *atomic actions*, section 2.2.3. This model *does not* include provisions for modelling fault tolerant systems, rather it describes the reliability of a base system, and requires extension to model fault tolerance.

This work concludes with a discussion of *reliability growth* phenomena.

There is, therefore, no direct correlation between the reliability model of Laprie & Kanoun, and the recovery block model presented in this thesis. Their model is, however, a significant source of inspiration for the model developed here, in particular for the application of a random fault model to a software based system.

6.4.5 Arlat et al.

The model of Arlat *et al.* [5] is a basic Markov chain model, with three final states: completed, detected (benign) failure, and hidden (catastrophic) failure. The process of execution of the recovery block, from an initial state, I, is modelled in a purely functional manner, with single states corresponding to the execution of the alternates, P and S1-2, and their acceptance tests, TP1-4 and TS1-4, together with benign, B, and catastrophic, C, failures. This is illustrated in figure 6.18 for a recovery block comprising a primary together with a single alternate.

Transition probabilities between the states of this model are based around the likelihood of independent and/or related faults occurring in the alternates. Timing of these faults is not considered. It is noted that the fault rates used in the model may be derived

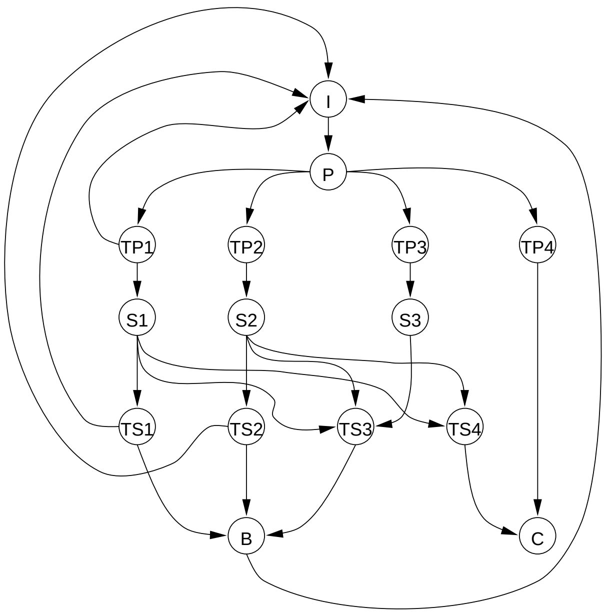

Figure 6.18: Arlat's Recovery Block Model

experimentally, in a similar manner to that envisaged for the estimation of the parameters of the model developed in this thesis.

Analysis of this model is based around the probability that the model enters one of the final fault states. The timing of system completion and/or failure is not considered.

Further sections of the paper consider N-version programming systems, nested recovery blocks, and the effects of unreliable acceptance tests.

This work is a good example of the use of a basic Markov chain model, which is applied to modelling the functional correctness of a recovery block system. The model is small, and easily analysed without the aid of tools; hence being simple to apply. The results obtained are of some applicability, but are necessarily limited by the omission of timing properties.

6.5 Other System Models

This work has been focused, primarily, on reliability modelling of recovery block systems. This section provides a brief description of the process by which the modelling techniques developed may be applied to other systems. A detailed analysis is not provided, since this is expected to be similar to that used for recovery block systems. Rather, the structure of the models is presented: from this, analysis is a straight-forward manner.

6.5.1 N-version programming Model

N-version programming was discussed in section 2.2.1: it is the software equivalent of N-modular redundancy in hardware design. In a similar manner to the recovery block, N-versions of a software module are derived: unlike the recovery block, those modules are then executed concurrently, with the same inputs, and a voter then compares the results. The result produced by the majority of the versions is passed as the result of the entire N-version programming system.

It is clear that the versions in an N-version programming system correspond to the alternates in a recovery block. In both cases, diverse design is used to achieve reliability. The difference between the two systems lies in the sequential execution of the recovery block, compared to the parallel execution of versions in N-version programming . Because of this similarity, it is possible to model the diverse versions in an N-version programming system in a similar manner to that used for a recovery block, with the only difference being due to the use of a single voter compared to multiple acceptance tests.

A model for an N-version programming system may therefore use the generic system model described in chapter 5 as the basic model for the behaviour of the versions. This leads to a model structure, for a 3-version system, as illustrated in figure 6.19. Despite the obvious similarity of this model to the recovery block model, there is a major difference in the structure of the voter, as compared to the acceptance test model used previously. Once again, the voter is assumed *infallible*, and maps from correct execution of the versions to a final *pass* state, and from incorrect execution to a final *fail* state. The difference here, is that a synchronisation point is introduced, ensuring that results are not produced until all versions have completed their execution.

An N-version programming system with a fallible voter may be modelled by additions similar to those used to model a recovery block system with fallible acceptance test. Additional transitions are added from the faulty state of each version to the pass state of the entire system, and also from the completed states of each version to the final system fail state. Again, the resulting model is relatively simple, and may be analysed in a similar manner to that used for recovery block systems.

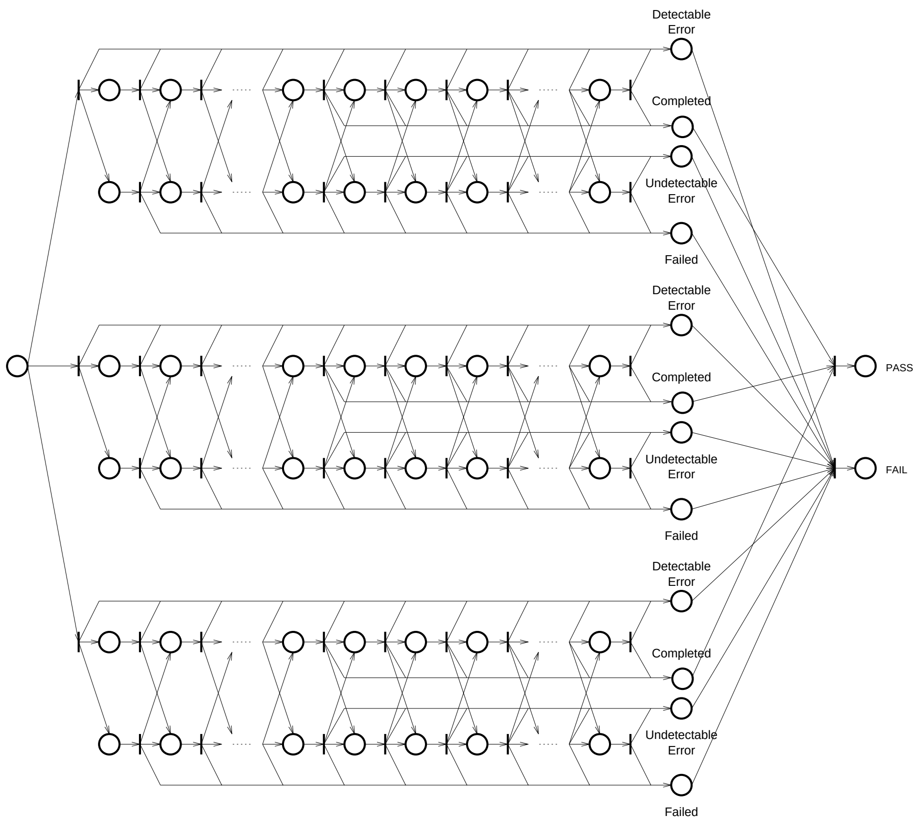

Figure 6.19: Initial N-version programming model

6.6 Summary

This chapter has discussed the application of the generic real-time system reliability model, developed in chapter 5, to the modelling of fault-tolerant systems. It has been shown that the results obtainable offer new insights into the behaviour of fault tolerant systems, and will benefit the designers of such systems.

Fault tolerant systems are built in a hierarchical manner: at the lowest level, these systems comprise a large number of fault intolerant building blocks. These building blocks are combined, using techniques such as those discussed in chapter 2, to provide fault tolerant modules, each of which performs a single aspect of the operation of the entire system. A number of these modules combine to form the complete system.

The generic real-time system reliability model, developed in chapter 5, cannot be used to directly model the behaviour of most complete fault-tolerant systems: to do so would be a significant oversimplification. It is, however, useful for modelling the behaviour of small, clearly defined, pieces of a system, with definite start and finish points, and clear failure modes. As such, it maps cleanly onto the lower levels of a fault-tolerant system, as a technique for modelling the behaviour of the basic building blocks from which the entire system is built. This process was discussed in more detail in section 6.1.

Once these building blocks have been modelled, it is necessary to model the combination of several such blocks into a single fault-tolerant module. There are many ways in which these blocks may be combined to achieve fault tolerance, of these the recovery block and N-version programming schemes have been the most widely studied. The work presented in sections 6.2 to 6.4 provides a further study of the recovery block technique, using the models developed in this thesis.

In section 6.2.1, the basic properties of a recovery block were studied. It was shown that the model developed in this thesis permits for modelling both the reliability and timing properties of the recovery block. If was further shown that a recovery block system has two modes of operation: with low failure rates, the individual alternates dominate, and the system shows a number of likely completion times, based on the run time of the alternates. As failure rates increase, the recovery block shows more homogeneous behaviour, with a range of completion times, and the effects of the individual alternates being much less visible. In addition the recovery block failure rate increases dramatically, and failures occur earlier.

These effects are important, since they show that the timing properties of a recovery block are greatly affected by the order of execution of the alternates within it. This effect has not been considered in previous recovery block models, since the overall system reliability is not affected by this process. It is clear, however, that if completion time, in addition to reliability, is important, the order of execution of the alternates within the recovery block must be considered carefully. It has been shown that the reliability modelling techniques developed in this thesis are capable of providing sufficient reliability and timing information for this purpose, and a number of examples of this process have been presented.

Results for recovery block systems with fallible acceptance tests have also been presented. These fallible acceptance tests are shown to have the potential to produce systems which appear more reliable than they should. Misclassification of faulty results as correct will result in systems which appear to have completed successfully, but in fact have hidden faults contained within them. The model presented herein allows this to be modelled using simple parameters, and in addition provides an estimate of a system's reliability with a perfect acceptance test, from which some idea of the reliability of the acceptance test in use may be determined.

This chapter has concluded with a brief discussion of the means by which an N-version programming system may be modelled, based again on a combination of building blocks each modelled using the generic system model. It is clear that this technique is applicable for such systems, and should be extensible to other techniques for building fault tolerant systems.

It is clear that the model developed in this thesis performs better than other reliability models developed in the literature, since it allows for both the reliability and timing properties of a system to be modelled. Given this information about the timing properties of a system, the designer is in a better position to determine the scheduling for that system. The application of these this information to enhancing scheduling decisions is discussed in chapter 7.

Chapter 7

Application to Scheduling

In chapter 6, the modelling of fault-tolerant software systems, such as recovery blocks, was discussed. It was shown that the techniques developed in chapter 5 are suitable for the modelling of such systems, and that these techniques allow for both the reliability and timing properties of such systems to be derived. The results of this analysis comprise plots of the completion and failure probabilities of the system against time, together with the values of a number of common metrics, such as, for example, mean completion time. It is the provision of this detailed timing information, in addition to the reliability data, and coarse grained timing metrics, which sets this model apart from other system reliability models.

In this chapter, a number of techniques which may be applied to the scheduling of real-time, fault-tolerant systems are discussed. It will be shown that, with the provision of detailed timing information, it is possible to derive schedules for the execution of a system, which are more efficient than those which may be derived in the absence of this information. This has been a further motivation for the design of the model developed in this thesis: the belief that the coarse-grained timing metrics provided by many system reliability models lead to inefficient scheduling, and much waste of resources.

7.1 Problems in Real-Time Scheduling

In the taxonomy of Casavant & Kuhl [16], reproduced in figure 7.1, the range of scheduling possibilities for a distributed computer system is shown. From this, it is clear that there are a number of choices which must be made when selecting a scheduling policy for a real-time, fault-tolerant system: In particular, the choice of *static* versus *dynamic*

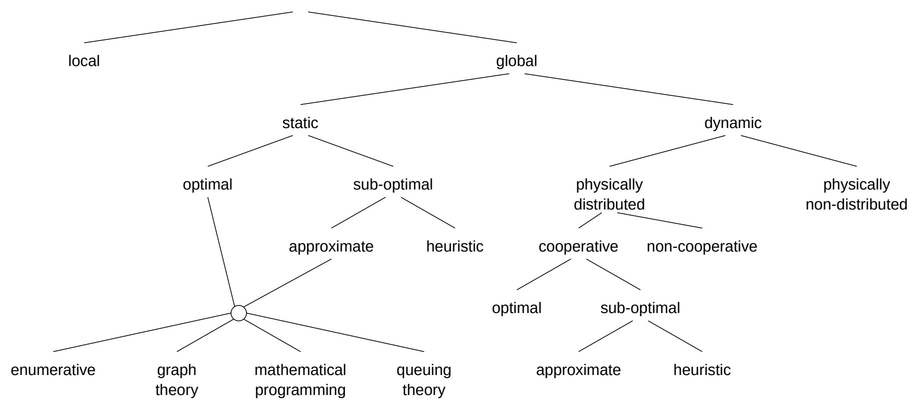

Figure 7.1: The scheduling taxonomy of Casavant & Kuhl

scheduling, and between *optimal* and *sub-optimal* algorithms is of great importance.

In Casavant & Kuhl's work, the choice of static versus dynamic, reflects the assignment of processes to processors. The term is used here in a broader sense: not only are processes assigned to particular processors, when static scheduling is used, but the scheduling of those processes on that processor is also fixed. Alternatively, a dynamic scheduling scheme may not only assign a process to a different processor each time it is scheduled for execution, but will also affect the scheduling order of processes on a particular processor. In the field of scheduling for safety critical, embedded systems, the usual choice is static scheduling. The potential for failure with dynamic scheduling schemes is usually considered too great.

The work of Xu & Parnas [93] considers this in more detail, and strongly recommends the use of *pre-run-time* scheduling, possibly the strongest form of static scheduling. The importance of static scheduling, where a schedule may be constructed to ensure that a set of processes will always meet their deadlines, is underlined, when it is stated that

> "Currently, many safety-critical hard-real-time systems are built using methods which do not provide any guarantee that critical timing constraints will be met. As a result, timing errors where computations miss their deadlines are the most unpredictable, most persistent, most costly, and most difficult to detect and correct type of errors in safety-critical hard-real-time software."

and further

> "For satisfying timing constraints in hard-real-time systems, predictability of the system's behaviour is the most important concern; pre-run-time [static]

> scheduling is often the only practical means of providing predictability in a complex system."

To ensure reliability, it is seen to be vital that the process schedule is derived in advance.

The choice of optimal versus sub-optimal algorithms is more difficult. Casavant & Kuhl, discussing general purpose systems, state that:

> "In the case that all information regarding the state of the system as well as the resource needs of a process are known, an *optimal* assignment can be made based on some criterion function. Examples of optimisation measures are minimising total process completion time, maximising utilisation of resources in the system, or maximising system throughput. In the event that these problems are computationally infeasible, *suboptimal* solutions may be tried." [16]

In the case of safety critical embedded systems, the choice of optimal versus sub-optimal is less clear cut. In some areas, performance is of vital importance, and hence an optimal solution should be derived if at all possible. In other areas, a suboptimal solution, as measured by a particular metric, may be required to ensure safety, even if it is possible to derive an optimal solution.

Indeed, much of the work on scheduling real-time systems has concentrated *entirely* on suboptimal algorithms. In the field of embedded control systems, it is typically considered more important to ensure safety than to achieve maximum performance. In cases such as this, the precise details of a processes execution time may be neglected in favour of a worst-case estimate. Indeed, the precise timing properties of a system may not always be predictable, yet it is still possible to derive safe schedules using a pessimistic estimate of the worst case execution time of a process.

This approach to scheduling makes a major assumption: that the worst case execution time of a process may be found. In certain cases, it is not possible to determine the worst case execution time, or if this worst case can be determined, it may be found to be much larger than the typical execution time. This is discussed by Haban & Shin [35] who state that:

> "Due to data-dependent loops and conditional branches in each program and resource sharing delay during execution, the [worst case execution time] is usually difficult to obtain and could be several orders of magnitude larger than the true execution time."

The results presented in chapter 6 of this thesis show the same effects: systems typically have a long-tail to their execution time distribution, and the majority of systems complete in a much shorter time.

These effects are further noted by Audsley *et al.* [7], who note that they cause severe under utilisation of resources in many systems. They note that this under utilisation comes from:

1. Software components not taking worst case execution paths.
2. Hardware behaving better than expected – due to gains from caching, pipelining or other facilities.
3. Sporadic processes not executing at their maximum rate.
4. Error handling software (eg: exception handlers or alternatives in a recovery block model) not executing.
5. Spare time incorporated by schedulability analysis to guarantee hard deadlines offline not being required at runtime.

Much of the under utilisation may be identified as *spare capacity*: processor time that is not required at run-time to meet deadlines of crucial tasks.

It may be seen, therefore, that there are two contradictory goals to be met in the scheduling of real-time, safety-critical embedded systems:

- *The desire for safety*: It is *vital* that safety is ensured at all levels. Engineers must strive at all times to design a system which is safe to the extent that is *reasonably practicable* [28].

- *The need for performance*: In many embedded systems severe cost, space, and weight constraints apply. In cases such as these it may not practical to have a system operating with much spare capacity, and some increased risk may have to be accepted to meet other constraints.

When coupled with the increasing drive towards greater use of software-based systems for embedded control, and the increasing functionality expected from those systems, it seems clear that the problem of scheduling such systems will not become easier. This problem has been summarised by Stankovic & Ramamritham [86], who note that:

> "The next generation of real-time systems will be large, complex, distributed, adaptive, contain many types of timing constraints, need to operate in a

highly non-deterministic environment, and evolve over a long system lifetime [...] for these systems, the typical semantics (all tasks making their deadlines 100% of the time) associated with [...] static real-time systems is not sufficient."

Given these problems, it seems clear that reliability modelling techniques should be extended to model the timing properties of fault-tolerant embedded systems, in order that reasonable design decisions may be made. Such modelling techniques may then be used as a design aid during schedulability analysis. In particular, they may be used to derive a schedule which takes into account the probability distribution of the system's execution time: allowing the reliability/performance trade off to be made explicit.

7.2 Application of Detailed Timing Data

As was discussed in chapter 6, the model developed in this thesis provides for determination of both reliability and timing properties of a system. This information may then to applied to scheduling analysis.

There are two ways in which this information may be used. The first is to use the full completion/failure time probability distribution in order to derive a probabilistic schedule which allows the system to meet all deadlines with a high probability. Determining such a schedule is a difficult problem: there are many variables to consider, and a large range of potential solutions. For all but the simplest systems, this is expected to be a computationally intractable problem.

The second method by which timing information may be used is to produce tighter bounds on a processes execution time. Many processes have a long tail to their completion/failure time probability distribution, and rather than derive a schedule based on the worst case execution time (allowing 100% of the components to complete execution before their deadline), it may be possible to use a reduced upper bound on process execution times, and allow a small fraction of the systems to exceed their deadlines. In many cases it may be possible to achieve large reductions in system slack-time, with only a small additional risk of failure.

Given more detailed timing information, and a knowledge of the expected use of a system, the designer is in a position to make an informed decision on whether the absolute worst-case execution time must be used, or whether a reduced set of bounds can be chosen, with a specific risk that the system will fail to perform within these bounds. In many cases the absolute worst-case behaviour is sufficiently unlikely, and the failure probabilities of other parts of the system are sufficiently large, that the increased probability of time-bound over-run will be tolerable.

The generation of schedules which allow all processes in a real-time system to meet all their deadlines all the time is not, in many cases, a feasible task. Once this is recognised, it is clear that reliability modelling techniques, such as those developed in this thesis, which provide detailed timing information, in addition to overall reliability data, must be utilised, in order to design systems which operate to a specific, tolerable, level of risk.

7.3 Summary

In this chapter, the problems inherent in the scheduling of real-time, safety-critical systems have been discussed. It has been shown that most current approaches to scheduling this class of system are based around coarse-grained, worst case, timing information. The result of this is the generation of inefficient schedules.

As systems become more complex, and more stringent performance demands are placed upon them, it is clear that these inefficient scheduling techniques will no longer be sufficient. The application of detailed timing information, should such information be available, may lead to more advanced scheduling techniques which make the trade-off between absolute safety and performance explicit. The development of system reliability models, such as that developed in this thesis, which provide detailed information on the timing properties of real-time, fault-tolerant, systems is seen to be of great importance.

Chapter 8

Conclusions

The initial chapters of this thesis focused on techniques for achieving reliability in embedded systems. The need for fault-tolerance, in addition to fault prevention, was discussed in chapter 1 and this led, in chapter 2, to a discussion of techniques by which fault-tolerance can be introduced into a system. These techniques will, if applied correctly, allow systems to be constructed which can tolerate the effects of a wide class of faults and operate within strict performance criteria.

Before such fault-tolerant techniques can sensibly be applied to a system, there is a need to determine the effects they have on the reliability and failure modes of that system. In particular, it is important that an accurate failure/reliability model is available during the design of fault-tolerant and safety critical systems, since it is paramount that the safety of such systems is ensured by all means reasonably practicable. There are two aspects to reliability modelling for safety-critical, real-time systems: derivation of the overall system reliability, and modelling the timing properties of the system. A number of existing techniques for reliability modelling were discussed in chapter 3, and it was shown that many existing reliability models have focused primarily on deriving the overall reliability of a system. Whilst this is important, a knowledge of the timing properties of a system is of equal value, and in many cases this is either ignored or given secondary treatment.

This is a major problem with current approaches to modelling real-time systems: It is usual for the timing properties of the system to be abstracted away in order to give each process a maximum execution time. Provided such a maximum time can be assigned, it is then possible to devise scheduling algorithms which, given sufficient resources, will ensure that all deadlines are met. These algorithms are pessimistic since they rely on the upper bound of a process' execution time, where as in real systems, the probability of errors occurring is low and the execution time of most processes is typically much less than the maximum. The system therefore operates with much slack-time, implying low efficiency but high reliability.

In chapter 4, a new technique for modelling the behaviour of real-time systems was derived. This technique is based primarily around a Markovian system model, with additions to allow timed transitions, synchronisation, and concurrency. This basic technique is applied, in chapter 5, to produce a model for the behaviour of a generic real-time system. This is a discrete time model with a lattice structure which models the progress of a computation from its initial state to one of several final states: completed, detectable fault, hidden fault or failed. This model allows for both the functional and temporal behaviour of a system to be represented in a single high-level model, and is derived from generic properties of real-time systems, hence being independent of any specific design/implementation technique for such systems. This is an improvement on traditional system reliability models which typically focus on functional correctness and do not adequately model the temporal properties of such systems.

Fault-tolerant systems are built in a hierarchical manner: At the lowest level, these systems comprise a large number of fault intolerant building blocks. These building blocks are combined, using techniques such as those discussed in chapter 2, to provide fault-tolerant modules, each of which performs a single aspect of the operation of the entire system. A number of these modules combine to form the complete system. The generic real-time system reliability model, developed in chapter 5, cannot be used to directly model the behaviour of most complete fault-tolerant systems — to do so would be a significant oversimplification. It is, however, useful for modelling the behaviour of small, clearly defined, pieces of a system, with definite start and finish points, and clear failure modes. As such, it maps cleanly onto the lower levels of a fault-tolerant system, as a technique for modelling the behaviour of the basic building blocks from which the entire system is built.

As an example of this, the generic real-time system reliability model developed here is applied, in chapter 6, to a study of the properties of recovery block systems. The recovery block is a simple, commonly-studied technique, which allows fault-tolerance to be introduced into a system. As such, it provides a good testbed upon which the properties of a new reliability model may be studied. Application of this new reliability model to recovery block systems shows that interesting results are obtainable, and that these results will be of use to the designers of fault-tolerant systems.

It has been shown that the model developed in this thesis permits for modelling both the reliability and timing properties of the recovery block. It was further shown that a recovery block system has two modes of operation. Firstly, with low failure rates the individual alternates dominate, and the system shows a number of likely completion times, based on the run time of the alternates. Secondly, as failure rates increase the recovery block shows more homogeneous behaviour, with a range of completion times, and the effects of the individual alternates being much less visible. In addition the recovery block failure rate increases dramatically, and failures occur earlier.

These effects are important, since they show that the timing properties of a recovery block are greatly affected by the order of execution of the alternates within it. This effect has not been considered in previous recovery block models, since the overall system reliability is not affected by this process. It is clear, however, that if completion time, in addition to reliability, is important, the order of execution of the alternates within the recovery block must be considered carefully: the model developed herein allows for comparison of the effects of differing alternate orderings, based upon test data, and this should ensure that failures, if they are going to occur, happen early. This gives a higher level recovery mechanism time to operate before a system's deadline is exceeded.

Results for recovery block systems with fallible acceptance tests have also been presented. These fallible acceptance tests are shown to have the potential to produce systems which appear more reliable than they should. Misclassification of faulty results as correct will result in systems which appear to have completed successfully, but in fact have hidden faults contained within them. The model presented herein allows this to be modelled using simple parameters, and in addition provides an estimate of a system's reliability with a perfect acceptance test, from which some idea of the reliability of the acceptance test in use may be determined.

Chapter 6 concluded with a brief discussion of the means by which an N-version programming system may be modelled, based again on a combination of building blocks each modelled using the generic system model. It is clear that this generic model is applicable for such systems, and should be extensible to other techniques for building fault-tolerant systems.

It is therefore clear that the model developed in this thesis performs better than other reliability models developed in the literature, since it allows for both the reliability and timing properties of a system to be modelled. As is discussed in chapter 7, if the probability distribution of the process' execution times is known, it should be possible to design a system which relies on this to attain much improved efficiency, whilst still managing to operate within a tolerable level of risk.

8.1 Suggestions for Further Work

The system reliability model developed in this thesis provides a basis upon which both the reliability and timing properties of a system may be calculated. Whilst initial results, from the study of recovery block systems, look promising, further work must be undertaken before the full range of the applicability of this technique is known.

An obvious extension of this work is to further study other fault-tolerant techniques. Application to N-version programming systems has been briefly discussed, and this work should be continued, since N-version programming systems are in common use. In addition, other techniques for fault tolerance may benefit from analysis, although the use of these other techniques is considered less widespread that the use of recovery block and N-version systems, and many of these other techniques can be considered as variants or combinations of N-version programming and recovery block methods.

It is noted that the mathematical framework developed in chapter 4 provides a richer set of semantics than was used in the modelling of recovery block systems. In particular, the concepts of synchronisation and N-step transitions were not utilised. The extension to N-version programming mentioned previously is an example of a situation where the synchronisation primitives of the model are required, and other concurrent/voting systems will similarly require this feature of the model.

The use of the N-step transitions provided in the model has not yet been investigated in any great depth. It is hoped that this feature of the model will allow for faster solution of certain reliability models, with multiple places being collapsed into one together with a number of N-step transitions. It not believed to offer any additional expressive power to the model.

A further validation of the model would be a comparison with results obtained from study of an actual system. The analysis performed to date has been theoretical in nature, and although results are presented in chapter 6 which show correlation between this model and other published models, a comparison with actual reliability data would improve confidence in this model.

This last point highlights a major failing in most reliability models published in the literature. Realistic comparisons between reliability models and actual embedded systems are difficult, and expensive to perform, and there is a severe shortage of data with which to validate these models. Unless this situation changes, the development of effective system reliability models will always be a hit-and-miss affair, based more in abstract theory than sound engineering practice.

The recovery block model developed in this thesis is heavily dependent on the reliability data gathered for the individual alternates. Gathering this data can be difficult, especially if the alternates are complex, and there are problems inherent in obtaining a statistically valid sample of the system's state space. This difficulty has been noted a number of times in the literature, for example Littlewood [54] states that

> "Issues of data collection remain problamatical in this area. The poor quality of most collected data attests to the difficulties which have been experienced."

and that

> "The chief technical difficulty arises from the requirement that the data must be collected in an environment which is typical of the use environment. If the real use data is not available, this means that random testing must be carried out with a probability profile representing the use environment.

The problem therefore, is the selection of the *probability profile* for the random testing such that a valid sample of the input space is made. It is clear that a uniform probability profile is likely to be somewhat unrepresentative of the use environment of a system, and some technique must be employed to determine a more realistic probability profile. As an example of this, Geist *et al.*[30], propose the use of mutation analysis [22] to derive test cases. This is clearly not the only possible solution: techniques such as simulated annealing, genetic algorithms, monte-carlo simulation, *etc.* have also been proposed for this role.

The selection of test cases is clearly an area worthy of further study. At present, though, it is clear that extreme care must be taken to ensure the validity of test results, and hence the validity of the results produced by *any* software reliability model.

8.2 Summary

In this thesis, the reliability modelling and analysis of real-time, fault-tolerant, embedded systems has been considered. It has been shown that many existing reliability modelling techniques are inadequate for this task, since they model only the overall system reliability, and the timing properties of the system are either neglected, or reduced to simple metrics. A new reliability model has been derived, which permits the modelling of both overall system reliability, and the timing distribution of system completion and failure. This model is based on a set of high level system attributes, which it is expected are estimateable from experimental data. This model has been applied to the study of recovery block systems, and it has been shown that the results obtained are compatible with, and extend, a number of other system reliability models. The thesis has concluded with a discussion of the application of more detailed timing information to the scheduling of safety-critical real-time systems, where the trade-off between performance and safety is discussed. It has been shown that the additional information available with models such as that developed herein, allows designers to make informed choices regarding this tradeoff. Hence, system design to a specific, tolerable, risk level is enhanced.

Appendix A

Simulation Software

The system reliability model presented in this thesis requires automated analysis. In this appendix the tools developed to enable this analysis are described. These tools have been written from scratch, in a mixture of the Sather [70] and Tcl [71] programming languages. Since these programming languages may be unfamiliar to some readers, a brief overview of their facilities is now provided.

Sather is an object-oriented language which supports efficient computation, with powerful abstractions for encapsulation and code reuse, and constructs for improving code correctness. It is strongly typed; supports multiple inheritance with explicit sub-typing, which is independent of implementation inheritance; parameterised types; iteration abstraction (similar in some respects to the C++ standard template library); garbage collection; exception handling; assertions; pre- and post-conditions; and class invariants. The Sather compiler is freely available from the International Computer Science Institute, Berkeley, and provides an environment with performance comparable to that obtained using C++, with much reduced development costs. The Sather system is distributed under a generally unrestrictive license, however since a number of classes developed by ICSI are included within the simulation software developed for this project, the following notice is required:

> "This program is based in part on Sather libraries distributed free of charge by the International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley, CA 94706 and which may be obtained by anonymous ftp from icsi.berkeley.edu."

The Tcl system was developed by John Ousterhout, and is distributed by Sun Microsystems. It provides a high level scripting language, designed primarily for controlling small, semi-independent, software tools. Tcl is small, powerful, and easily extensible. It is this extensibility which has prompted its use in this project: Modules to perform numeric computation are written as high-performance Sather code, and then linked into the Tcl interpreter, where they become available as regular Tcl commands. The framework within which simulations are performed is driven through simple Tcl commands, providing a simple means of running multiple simulations, and of selecting parameters; and portions of the code which perform numeric computation are written in Sather to ensure high-performance.

It is worth commenting on the means by which these two systems are combined. The Tcl system is available as a set of C language library routines, which together form an interpreter. This interpreter evaluates commands provided as C language strings, whether taken interactively from a terminal, or provided from within another section of the program code. The ICSI Sather compiler does not produce native code directly, but rather compiles to C, and then automatically invokes the native C compiler. Sather provides the means by which a stub-class may be defined, with methods being coded directly in C code. This was primarily intended as a means by which access to low-level system libraries could be provided, but it is a simple matter to include the Tcl interpreter in a Sather program using this mechanism.

The two languages are not, however, a seamless fit: in particular, although both Sather and Tcl support exception handling, they do so in different ways, and it is not possible to cleanly map exceptions between the two systems: The Sather system allows exceptions to be arbitrary objects, Tcl restricts them to being strings. Whilst it is possible to generate a string representation for the type of a Sather exception, and to catch this from a Tcl script, the possibility of Sather exceptions returning arbitrary data in addition makes this issue difficult to resolve with any generality. For the purpose of this project this is not a significant restriction: Tcl is used to perform high-level scripting only (essentially iterating through several different parameter values, and directing output), so mapping Sather exceptions onto a single Tcl error return suffices.

These two programming environments are seen to provide a reasonable fit, and although there are some problems inherent in this combination it does provide both high-level scripting, and high-performance; with few problems.

A.1 Additional Tcl Commands

This section describes the additional commands added to the Tcl system as part of this simulation. These are accessed via the `netsim` system described in section A.2.

The `generate` command produces a system model, and returns an identifier `model_name` by which it may be referenced later. This command takes one of three forms, depending on the type of network model to be defined: The `generate alt` form produces an alternate model, such as that described in chapter 5. This corresponds to the generic real-time system model.

```
generate alt -pd <failure_rate> -pf <failure_rate> -pr <failure_rate>
             -cp <completion_profile> -as <model_name>
```

The `generate altat` form produces an alternate model with an acceptance test. The `-pca` and `-pir` parameters may be omitted when specifying this form, and default to 1.0 in that case, this corresponds to the model shown in figure 6.1. If these parameters are specified the model generated has a fallible acceptance test and corresponds to figure 6.11.

```
generate altat -pd <failure_rate> -pf <failure_rate> -pr <failure_rate>
               -pca <failure_rate> -pir <failure_rate>
               -cp <completion_profile> -as <model_name>
```

The `generate rb` form produces a three alternate recovery block model, by combining three alternates. This corresponds to the model illustrated in figure 6.2, assuming the alternates specified have infallible acceptance tests.

```
generate rb with <model_name> <model_name> <model_name> as <model_name>
```

In addition, the `forget` command causes the system to discard a model definition.

```
forget <model_name>
```

The `load_naf` and `save_naf` commands are provided to load and save network models to/from data files. The file format is text based, and human-readable.

```
load_naf <model_name> from <input_file>
```

```
save_naf <model_name> as <output_file>
```

The `evaluate` command performs an analysis of a system model. The limit limits of the analysis are specified, together with the set of states to evaluate. Output is produced as a file, which is processed with the `extract` command.

```
evaluate -from <model_name> -tmin <time> -tmax <time> -o <output_file>
         -s <state> [-s <state> ...]
```

The `extract` command takes data calculated by the `evaluate` command, and extracts data for a single state. Output is to a file, and the format of that file is a series of pairs of numbers, giving time and probability. This file may then be directly plotted, using a tool such as, for example, `gnuplot`, or analysed further if desired.

```
extract -i <input_file> -o <output_file> -s <state>
```

A.2 User Reference

The simulation environment comprises a single executable, `netsim`, which comprises a Tcl interpreter with a number of additional commands added. The `netsim` system operates in a manner similar to the standard `tclsh` command interpreter, reading commands either from files specified on the command line, or from standard input. There is no graphical user interface provided with this system.

When started the `netsim` tool displays the following message:

```
Network Simulation Tool -- Copyright (C)1995, 1996 CS Perkins
  $Revision: 0.4 $
  $Date: 1996/09/26 00:53:56 $
Instantiating Tcl interpreter and objects...
Adding new commands...
>
```

If a number of file names are given on the command line, the system proceeds to read command from those files, one-by-one; and exits when complete. If no parameters are specified, the system reads commands interactively. A typical command sequence is shown below:

```
set rb RB1
foreach pd {0.00050 0.05000} {
  generate altat -cp ../cp/alt1.cp -pd $pd -pf 0.001 -pr 0.001 -pca 0.99 -pir 0.8 -as alt1
  generate altat -cp ../cp/alt2.cp -pd $pd -pf 0.010 -pr 0.010 -pca 0.99 -pir 0.8 -as alt2
  generate altat -cp ../cp/alt3.cp -pd $pd -pf 0.010 -pr 0.005 -pca 0.99 -pir 0.8 -as alt3
  generate rb with alt1 alt2 alt3 as $rb
  evaluate -from $rb -tmax 100 -o $rb-$pd.apd -s 263 -s 264
  extract -i $rb-$pd.apd -o $rb-$pd.263.dat -s 263
  extract -i $rb-$pd.apd -o $rb-$pd.264.dat -s 264
  forget alt1
  forget alt2
  forget alt3
  forget $rb
}
```

This is the sequence used to generate the results for a recovery block with fallible acceptance test, used in chapter 6. As can be seen, most of the work is done using Tcl extensions written in Sather. The Tcl script itself provides simple scripting and parameter substitution only.

This then, is a brief description of the `netsim` system. At present this has system been run on Unix systems running NeXTstep, IRIX and Linux. Ports to other Unix systems should be trivial. For the purposes of producing the results illustrated in this thesis, running time of the simulations has not been excessive, even though the code utilises no special optimisation techniques: Most simulations were run on a NeXT computer, with 32MB memory, powered by a 25MHz Motorola MC68040 processor.

Appendix B

Generic Real Time System Model: Analysis

This appendix provides detailed results from an analysis of the generic real-time system model described in chapter 5 of this thesis. Three sets of results are presented, corresponding to the three standard systems described in section 5.3: binomial, exponential and uniform. For each system, plots of probability against time for each of the completed, detectable fault, hidden fault, and failed states are provided.

The results are shown in figures B.1 to B.12. These figures are presented in the form of intensity plots: darker colour indicates a point with greater probability. Exact numeric values are not shown, these results illustrate the shape of the plots only. The scale is preserved within each figure, but is not preserved between figures.

Each figure shows the results for a single one of the four final states of the model, for a range of values of the parameters p_d, p_f and p_r. Results are illustrated for a single overall completion probability only; it has been shown that the shape of the curves is the same for a range of different systems, provided the base completion probabilities follow the same pattern, so detailed results are omitted for the sake of brevity.

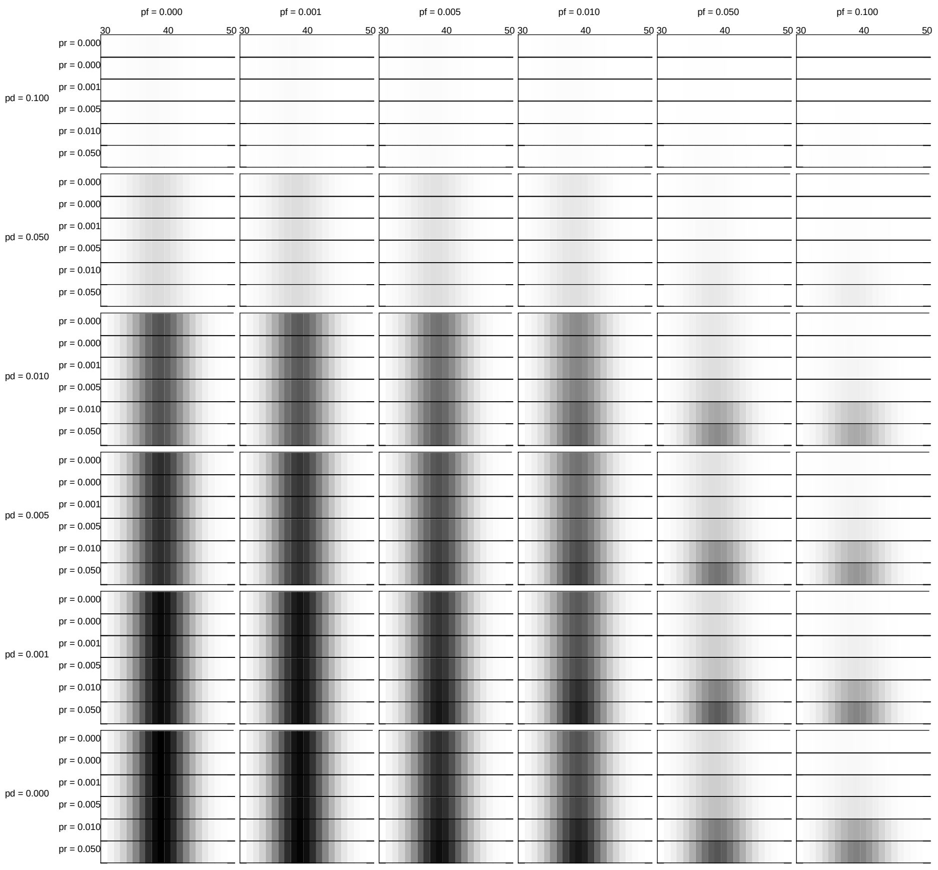

Figure B.1: Binomial 0.00100: Completed

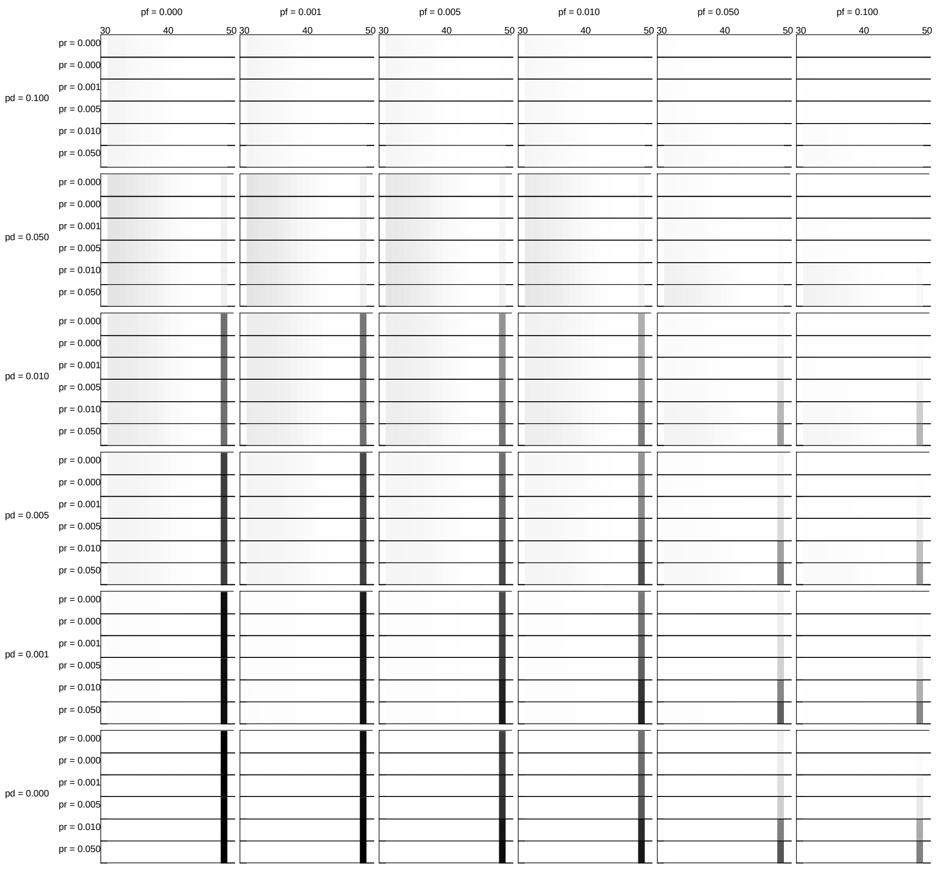

Figure B.2: Binomial 0.00100: Detectable Fault

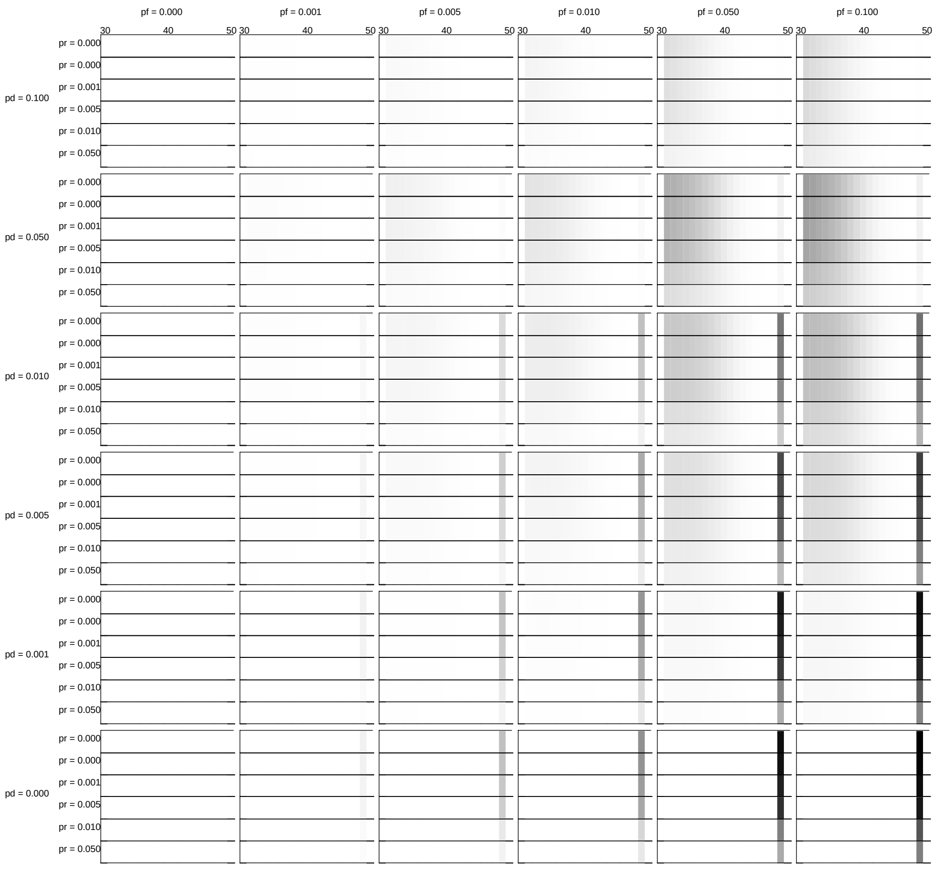

Figure B.3: Binomial 0.00100: Failed

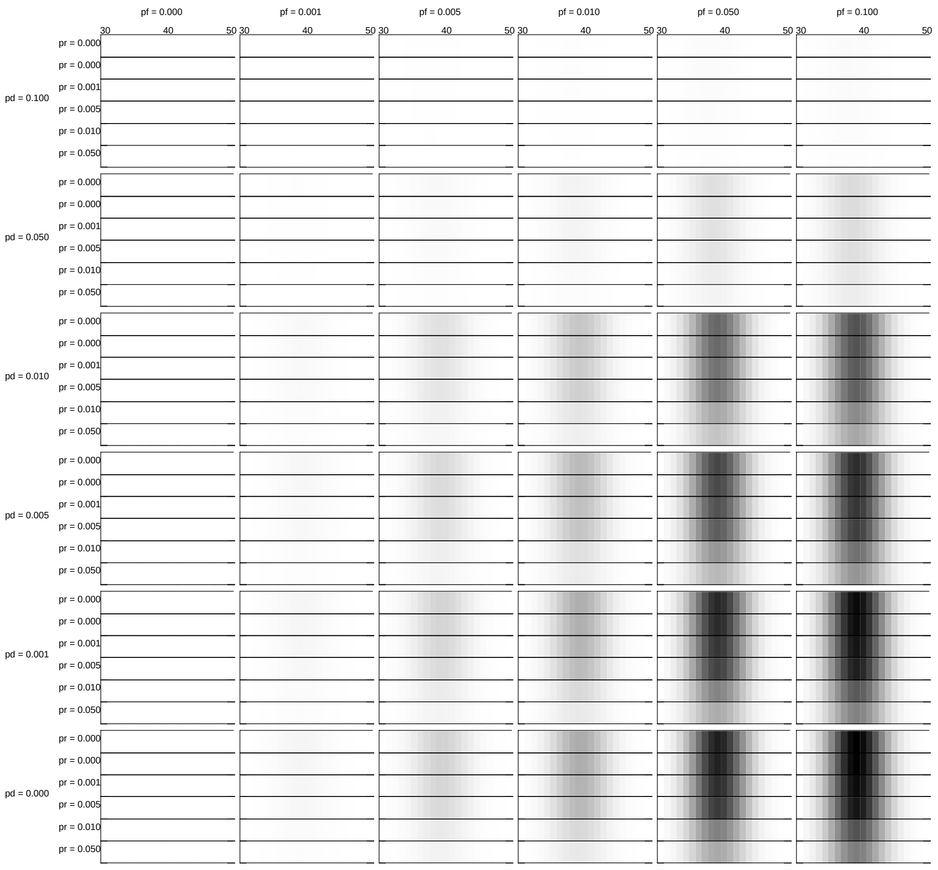

Figure B.4: Binomial 0.00100: Hidden Fault

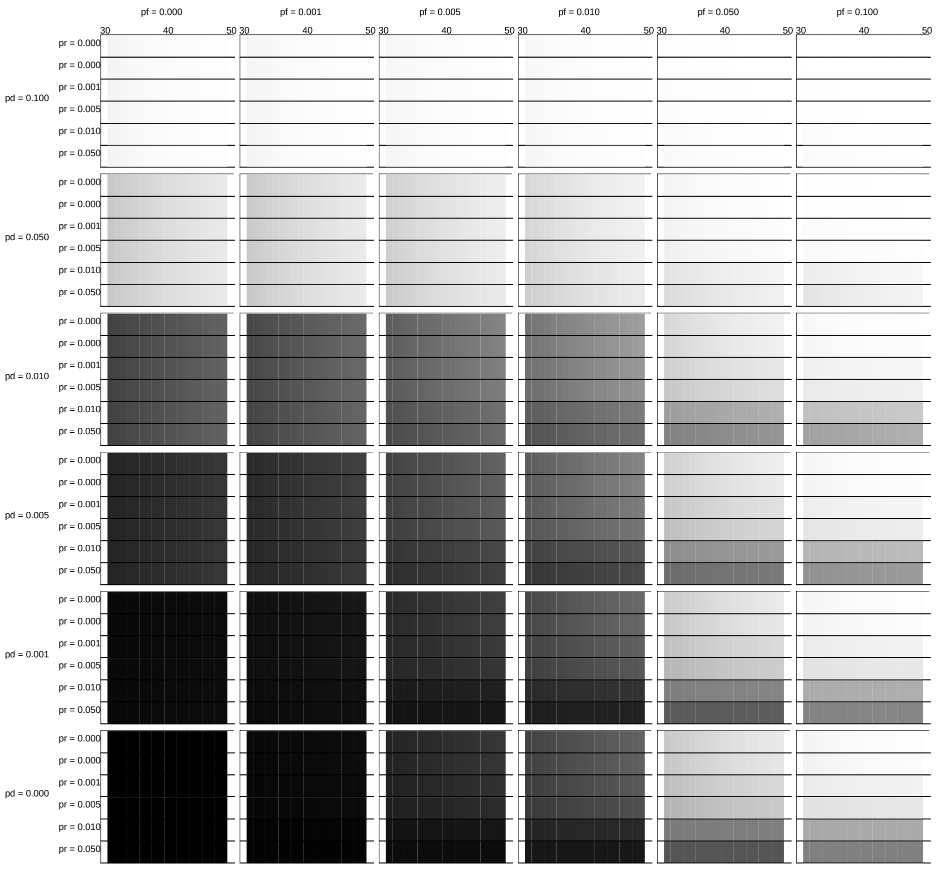

Figure B.5: Exponential 0.00100: Completed

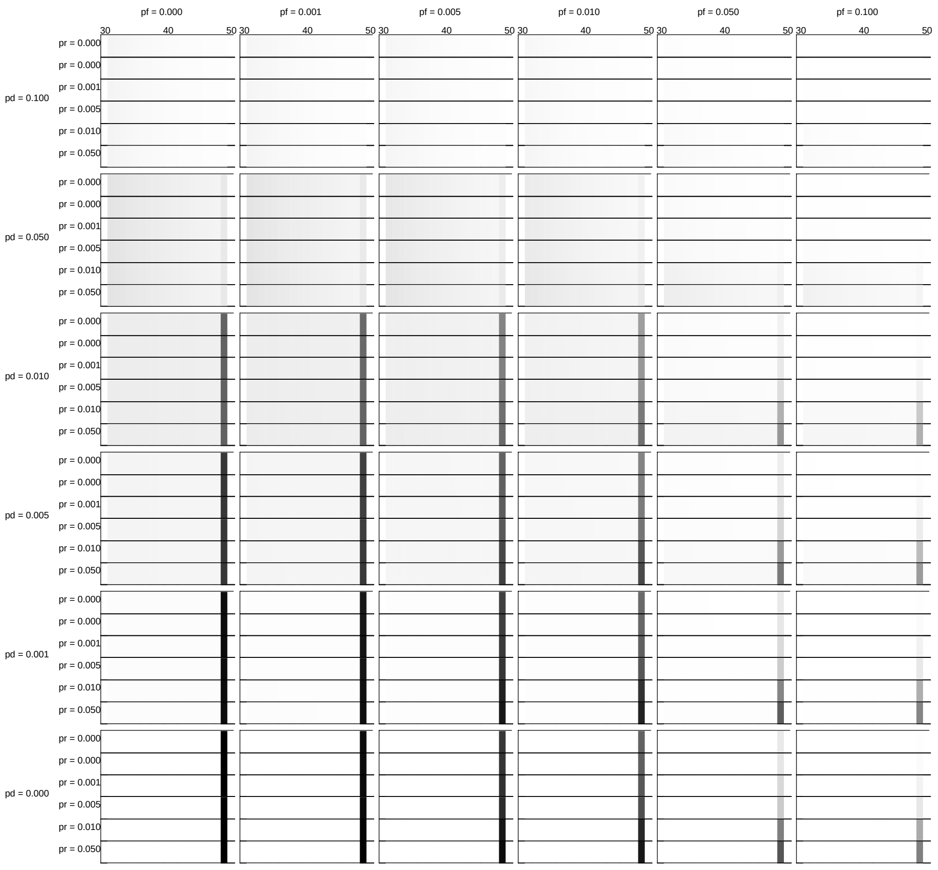

Figure B.6: Exponential 0.00100: Detectable Fault

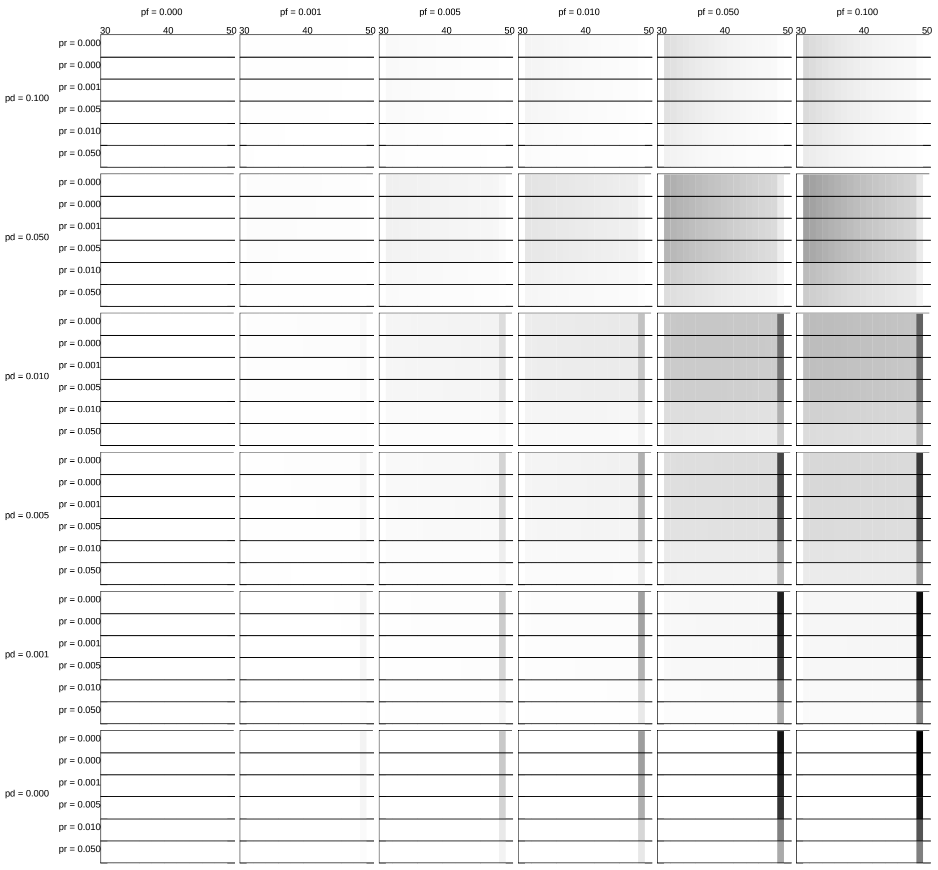

Figure B.7: Exponential 0.00100: Failed

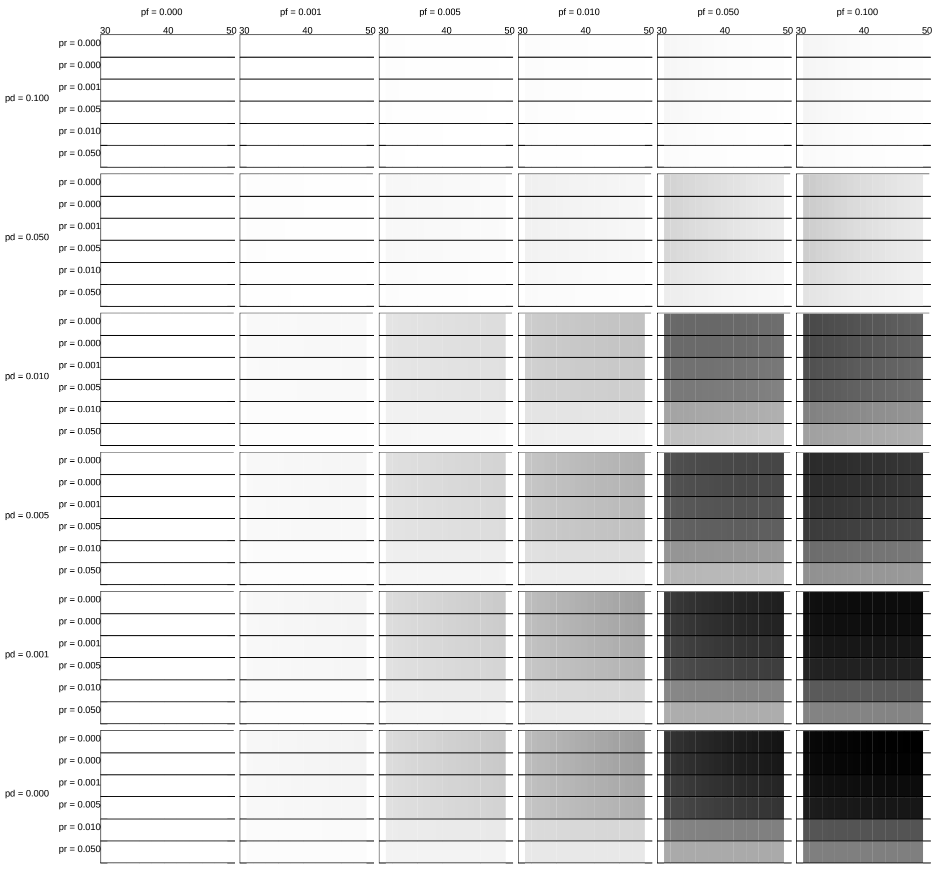

Figure B.8: Exponential 0.00100: Hidden Fault

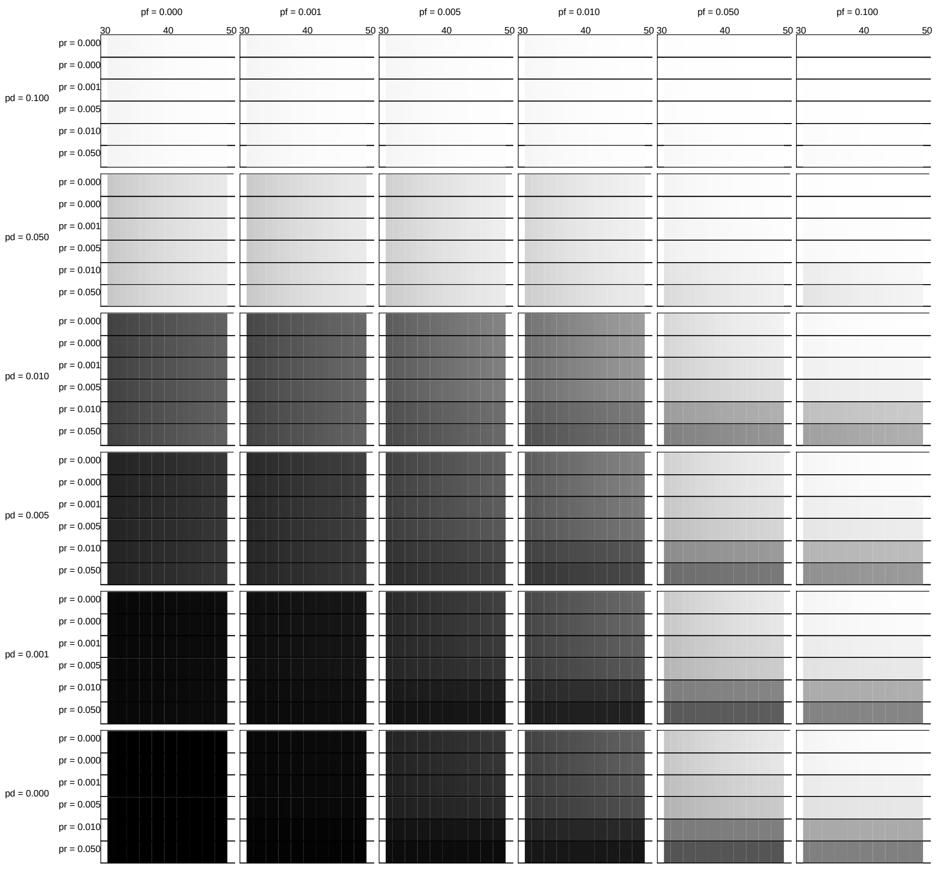

Figure B.9: Uniform 0.00100: Completed

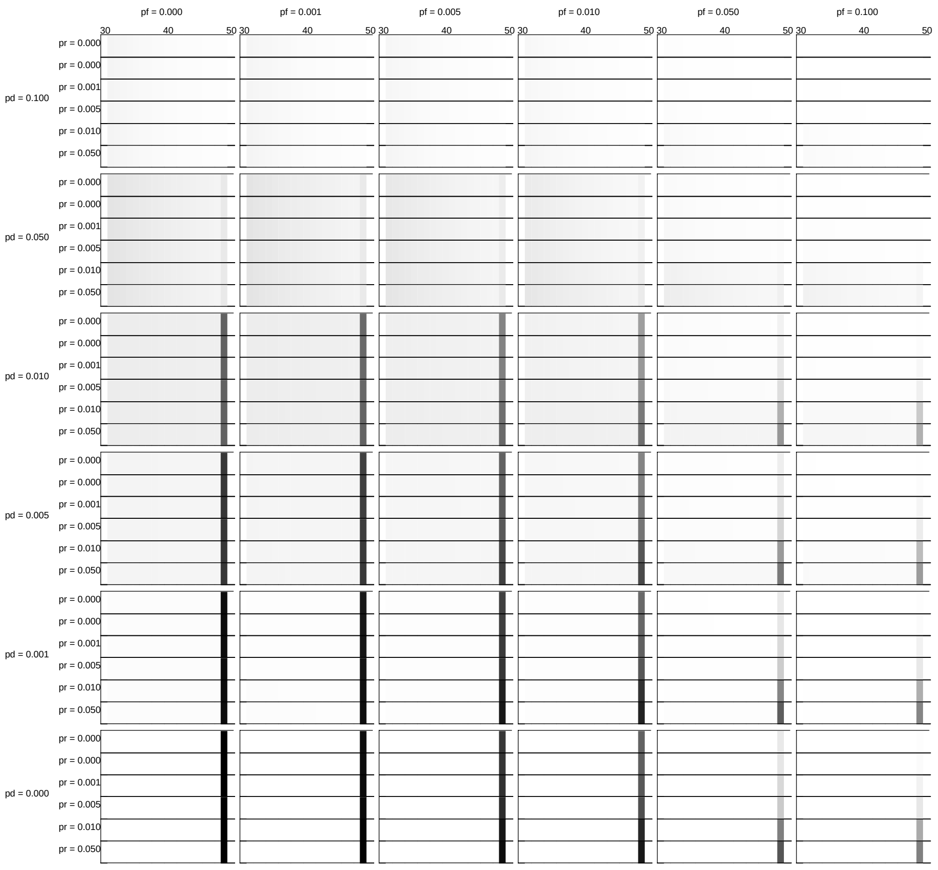

Figure B.10: Uniform 0.00100: Detectable Fault

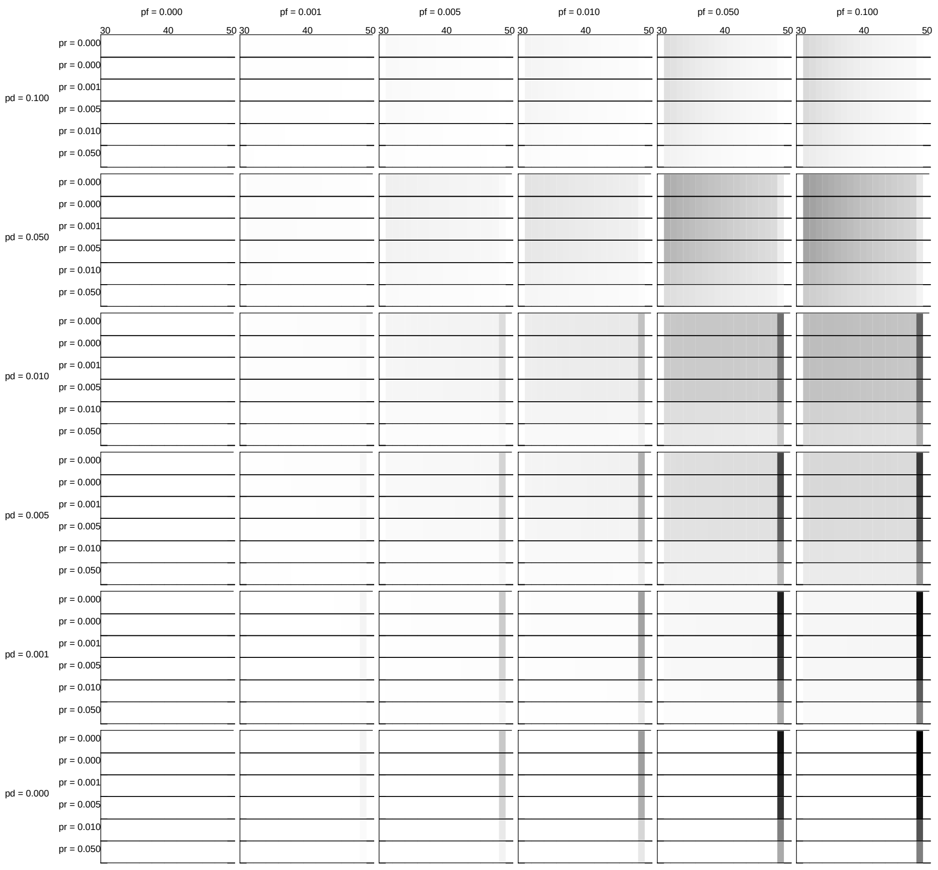

Figure B.11: Uniform 0.00100: Failed

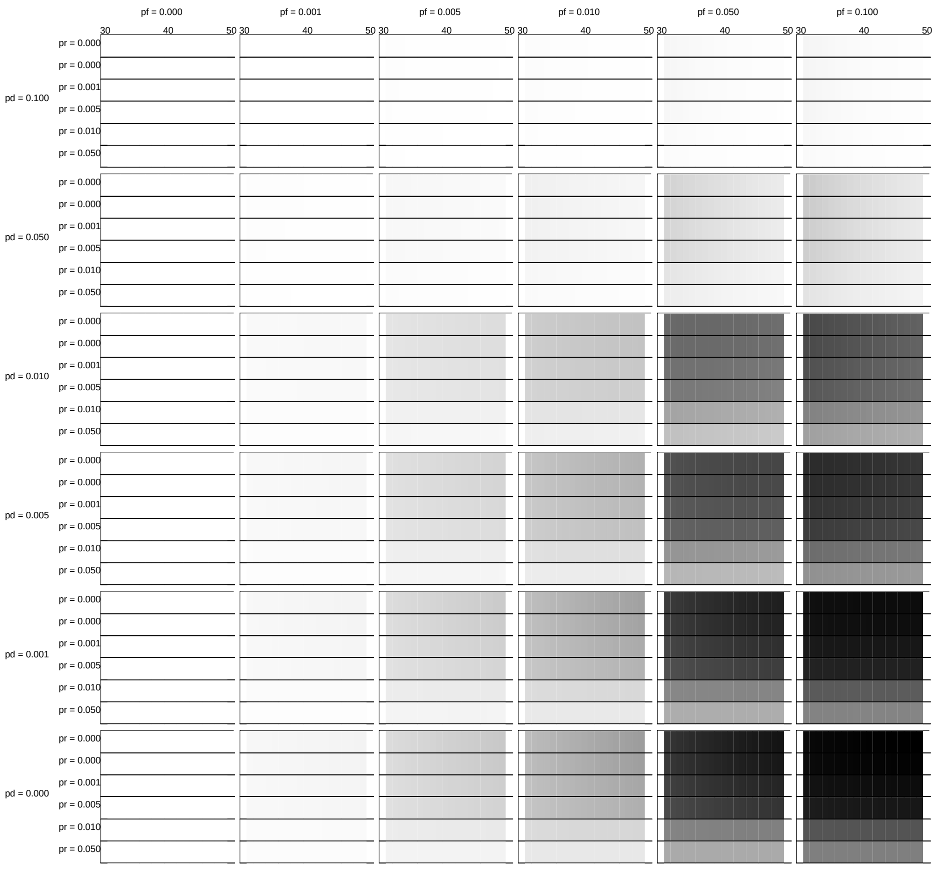

Figure B.12: Uniform 0.00100: Hidden Fault

Appendix C

Timing Properties of the Recovery Block

In chapter 6 a number of system reliability models were developed, and their behaviour studied. In particular, the effects of the alternate ordering on the behaviour of a recovery block was discussed in section 6.2.2. The purpose of this appendix is to provide additional data to further validate the conclusions drawn in section 6.2.2. This is achieved through the analysis of the properties of an two more recovery block systems: a modified version of the system studied previously, and a system comprised of a significantly different set of alternates.

C.1 Recovery Block System 1

The first additional system studied is based around that used in section 6.2.2, with the probabilities of hidden fault and recovery increased by a factor of ten, as in table C.1. The alternates used are unchanged, see figure 6.3 on page 65 for the completion profiles used. These alternates have been combined to form six recovery block systems, with the orderings chosen as in table 6.2 on page 68.

	p_f	p_r
Primary	0.010	0.010
1st Alternate	0.100	0.100
2nd Alternate	0.100	0.050

Table C.1: Alternate Parameters

Plots of completion/failure probability for these six recovery block systems is presented in figures C.2 to C.5. When compared with the completion profiles discussed previously, figures 6.5 to 6.8, it is clear that there are a number of features common to both systems: Although the exact value of the completion/failure probability is modified by the increased hidden fault/recovery rates in this new system, it is clear that the two systems show similar behaviour: the shape of the completion/failure probability curves is similar in both cases. As such, it is clear that the conclusions drawn in section 6.2.2 hold for this new system.

The mean completion and failure times for this new system are illustrated in figure C.1. When compared to the previous results, figures 6.9 and 6.10, it is once again seen that the results are similar. A greater number of systems show increases in their mean completion time in this new case, due to the differing failure probabilities; but the essential features of the curves remain constant.

It is clear, therefore, that a change in the model parameters does not affect the basic properties of a recovery block system, merely the details of its behaviour.

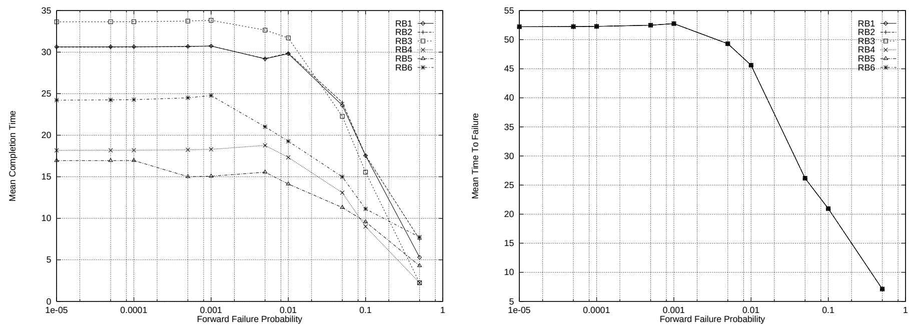

Figure C.1: Mean Completion/Failure Time

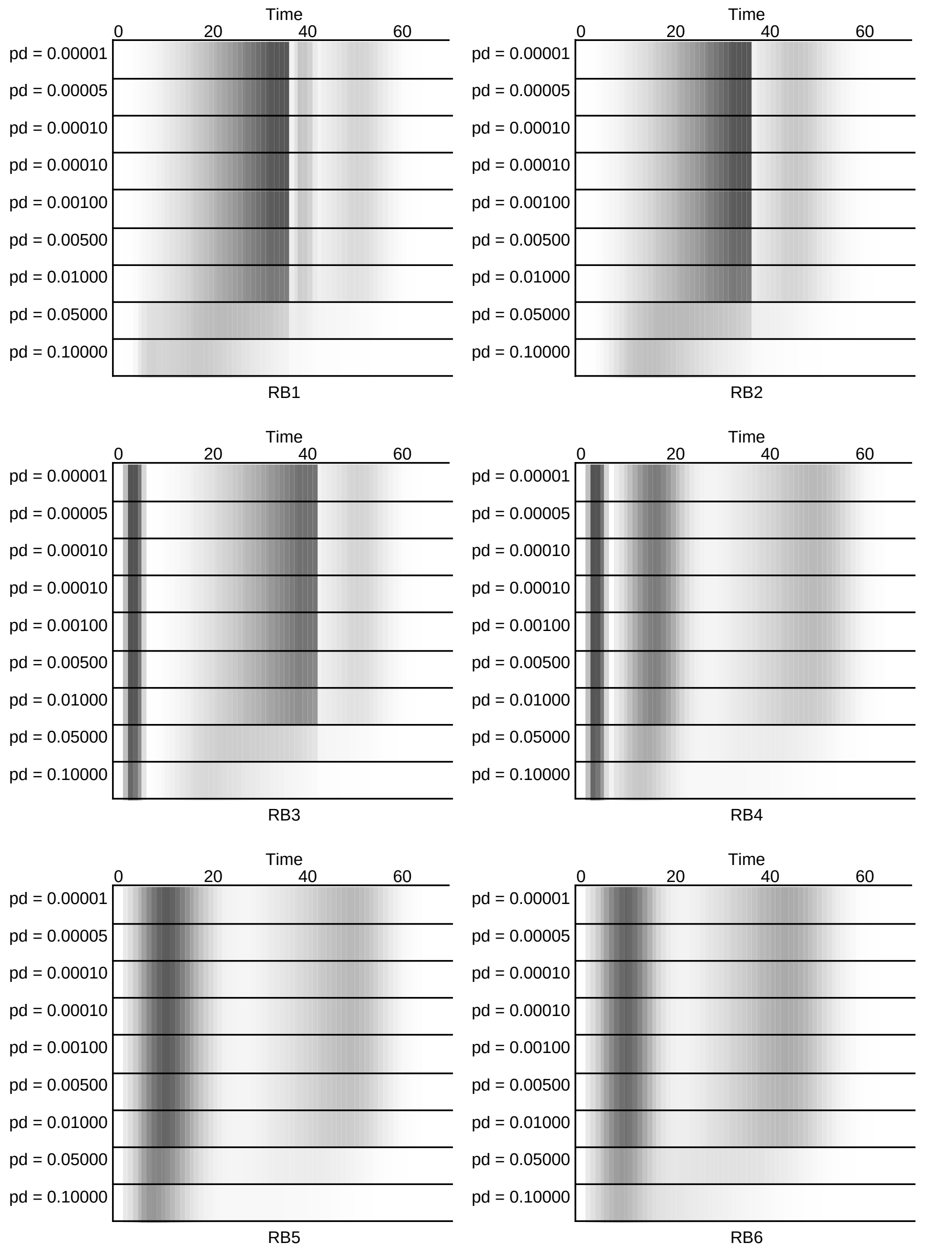

Figure C.2: Recovery Block Instantaneous Completion Probability for different p_d

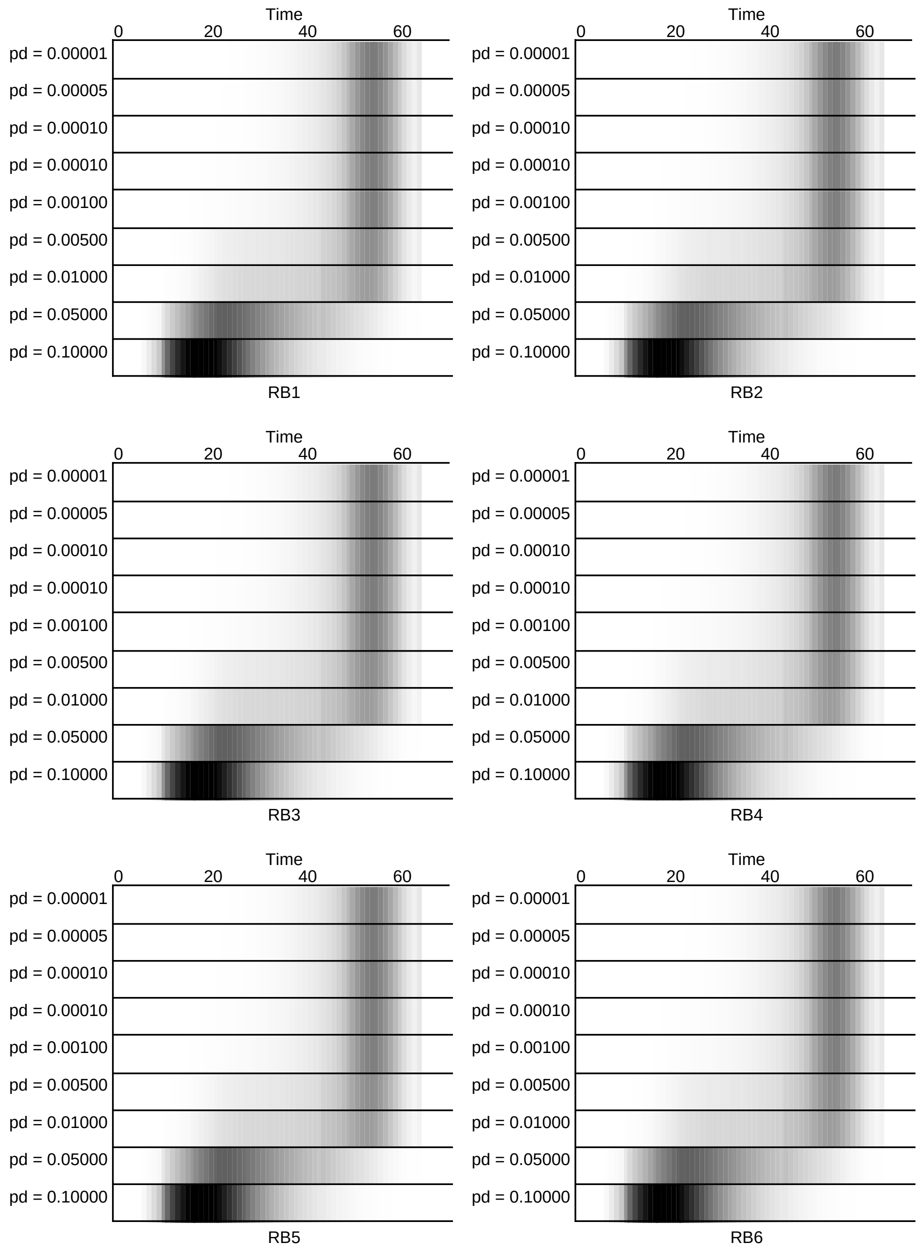

Figure C.3: Recovery Block Instantaneous Failure Probability for different p_d

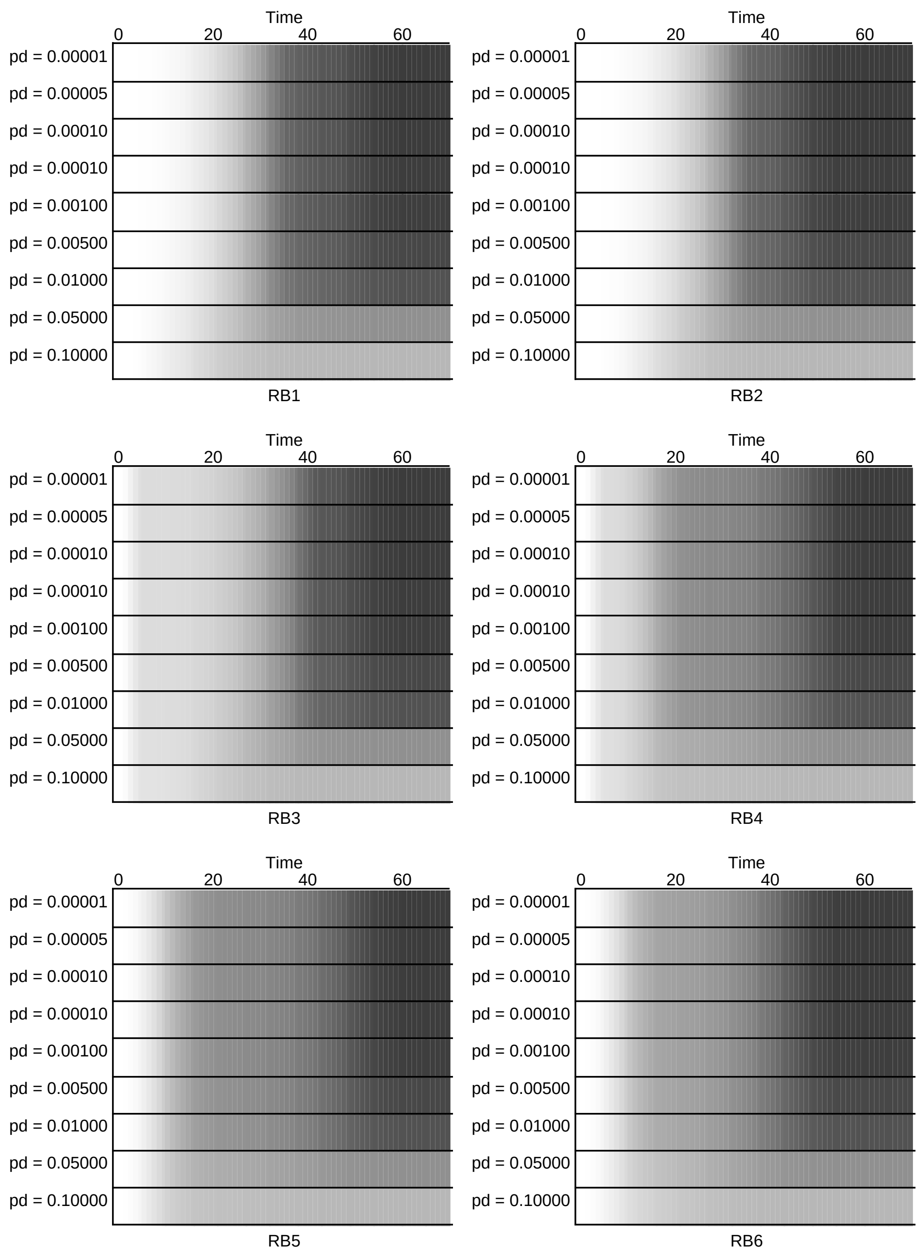

Figure C.4: Recovery Block Cumulative Completion Probability for different p_d

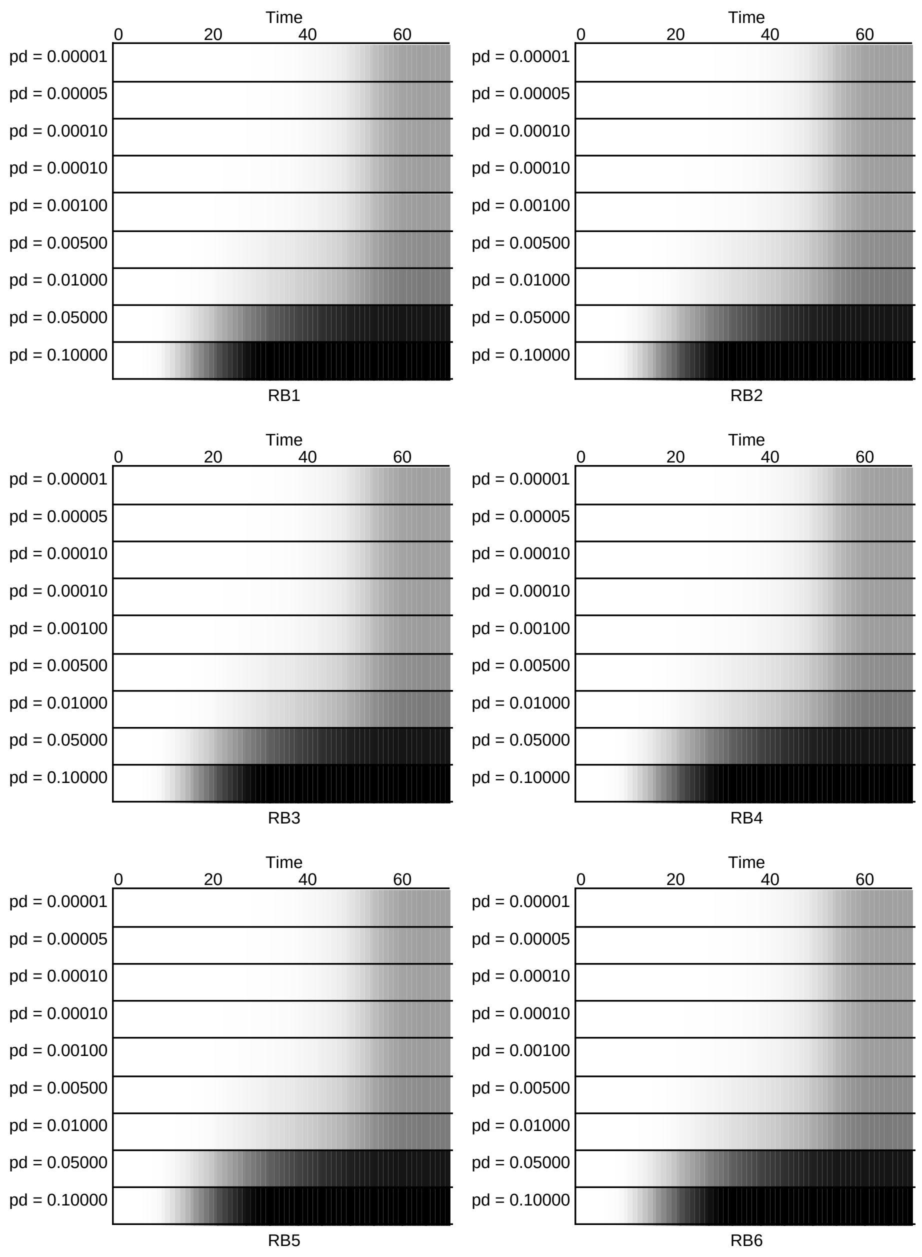

Figure C.5: Recovery Block Cumulative Failure Probability for different p_d

C.2 Recovery Block System 2

The second additional recovery block system has been chosen so that the alternates have significantly different completion profiles, than those used previously. The aim is to show that the properties of the recovery block noted previously are not an artifact of the particular set of alternates chosen, but generalise to a range of systems. This new set of alternates is illustrated in figure C.6. Once again, these three alternates have been combined to form six recovery block systems, with the orderings chosen as in table 6.2 on page 68.

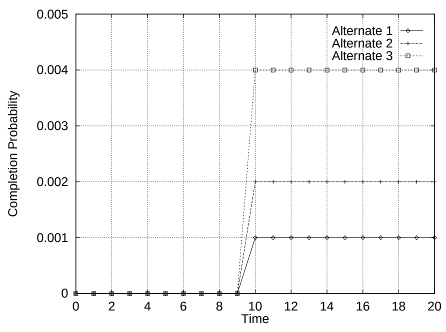

Figure C.6: Recovery Block Alternates

The plots of completion/failure probability against time produced by the simulation of these recovery blocks are illustrated in figures C.7 to C.10. When compared with the completion profiles discussed previously, figures 6.5 to 6.8, it is clear that there are a number of features common to both systems.

The instantaneous completion probability plots, figure C.7, show the alternates executing sequentially for small values of p_d; and as the forward failure probability is increased the three alternates become less distinct, and the completion profiles merge into one, shifting towards earlier failures.

The cumulative completion probability plots, figure C.8, show similar effects. As the failure rate increases the plots becomes smoother, with the effects of the individual alternates once again becoming less noticeable.

The failure profile plots, figures C.9 and C.10, are unaffected by the ordering of the alternates. As the failure rate is increased the failure profiles show the systems becoming more likely to fail, and that failures occur earlier during the execution of the system.

The mean completion and failure time data, figure C.11, also shows effects similar to those observed previously.

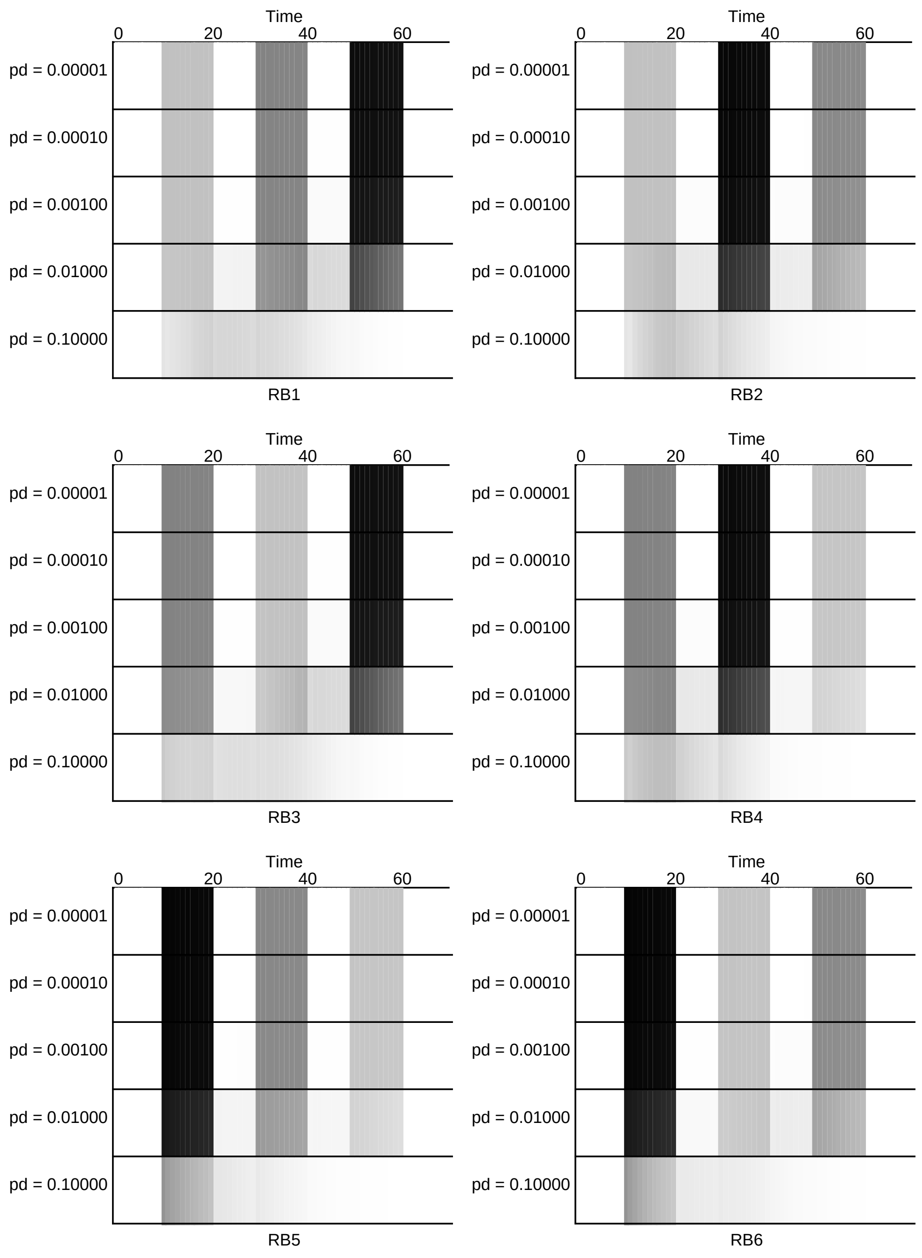

Figure C.7: Recovery Block Instantaneous Completion Probability for different p_d

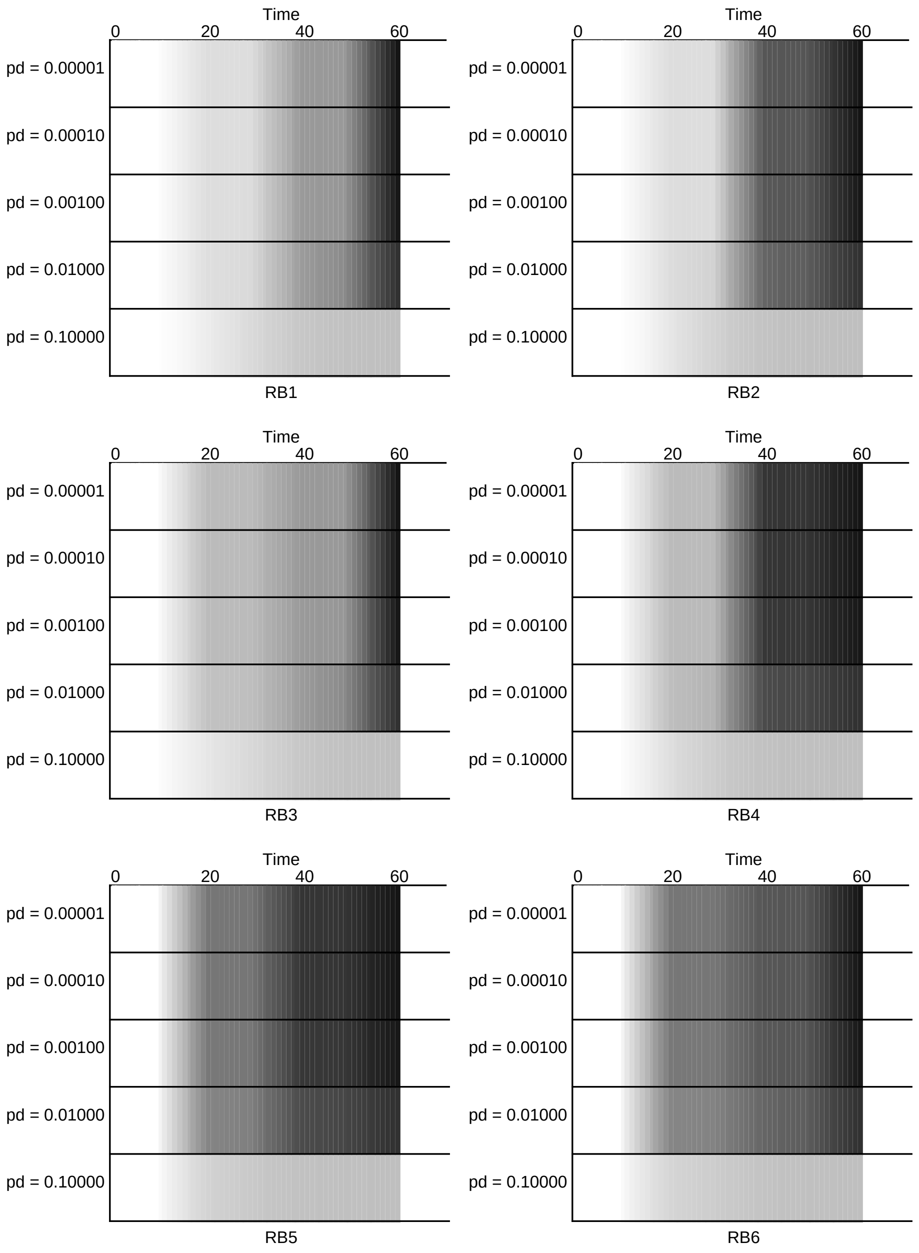

Figure C.8: Recovery Block Cumulative Completion Probability for different p_d

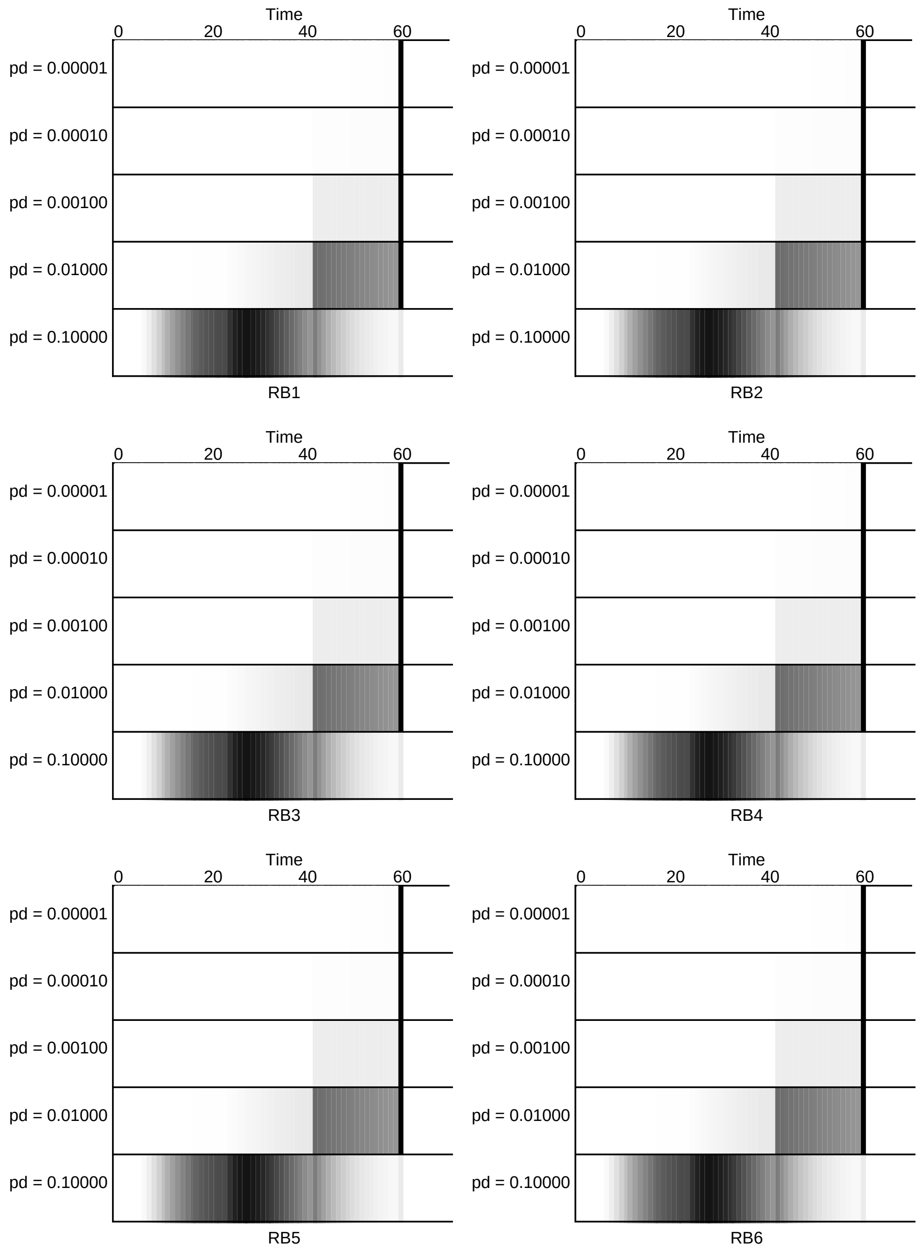

Figure C.9: Recovery Block Instantaneous Failure Probability for different p_d

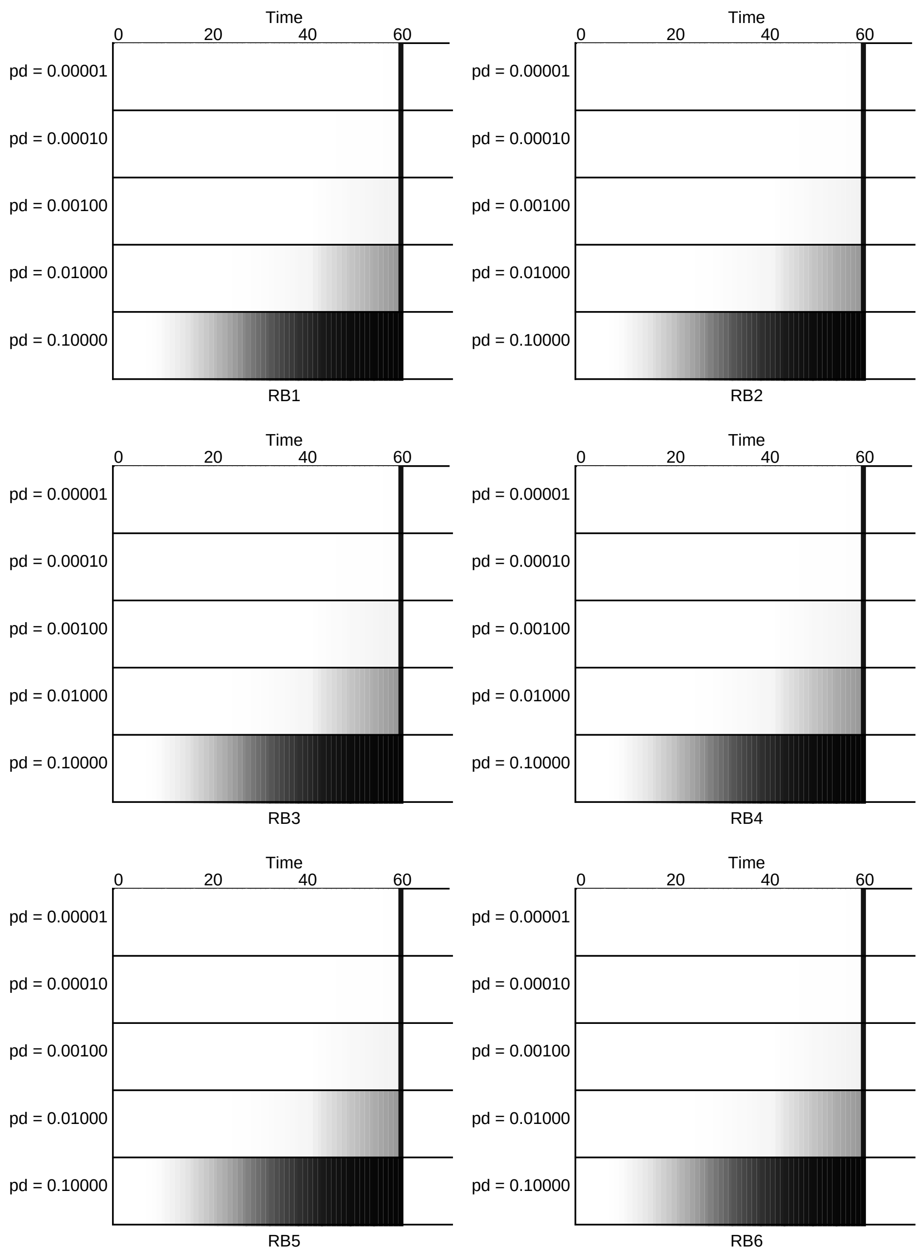

Figure C.10: Recovery Block Cumulative Failure Probability for different p_d

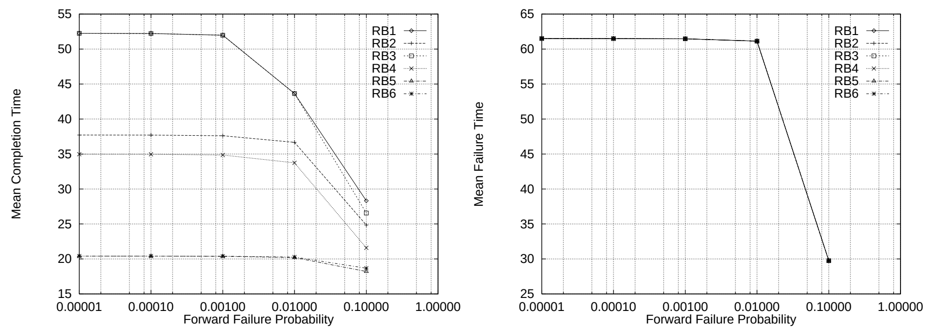

Figure C.11: Mean Completion/Failure Time

C.3 Summary

These results are intended to show that the properties of the recovery block are common across a range of alternates. As such, two systems have been studied in this appendix, and a further system in chapter 6. These systems have been chosen to have a range of properties, and hence it is hoped that the results produced, which are similar in many ways, illustrate this conclusion.

It is, of course, realised that a small sample such as this cannot provide proof that all recovery blocks operate in a similar manner. No study of the detailed timing behaviour of a recovery block can do that: the range of possibilities is too great. This study shows likely behaviour, and provides techniques by which the actual behaviour of a recovery block can be modelled based on the behaviour of the alternates comprising that recovery block.

Appendix D

Coincident Faults in Recovery Block Systems

As was discussed in section 6.2.3, it is not possible to assume that the alternates in a recovery block system fail in an independent manner. In this appendix an analysis of the recovery block model is performed, to illustrate the behaviour of this model with a number of dependent alternate failure modes. Three parameters of the recovery block model are candidates for modification due to the effects of dependent failure of the alternates. These parameters fall into two categories: those representing detectable faults, p_d; and those representing hidden faults, p_f and p_r.

In section D.1 the effects of simulating dependent alternate failure using increased values of the p_d parameter are discussed. This simulates the effects of increased detectable fault rates in the second and subsequent alternates in a recovery block.

In section D.2 the effects of modifying the second category of recovery block parameters are discussed. These modifications represent a recovery block system where dependent alternate faults cause the occurrence of faults which are not immediately detectable.

Results for the combination of these two effects are not presented here. These effects have been shown to combine in a linear fashion, and hence the presentation of detailed results for this would not be useful.

In this appendix the method of analysis, and the results obtained are presented. A detailed discussion of these results is presented in sections 6.2.3.1 and 6.2.3.2, and is not repeated here.

D.1 Effects of p_d

This section provides a detailed analysis of the recovery block system discussed in section 6.2.3.1. This system uses the same set of alternates, figure 6.3, and p_f and p_r parameters, table 6.1, as that described in section 6.2.2.

The effects of dependent faults in the alternates of a recovery block are here modelled by means of a *failure probability multiplier*. This is a small factor, p_{d_m}, by which the forward failure probability, p_d, is multiplied in subsequent alternates. That is, the forward failure probability in alternate n, p_d^n, is defined by equation D.1:

$$p_d^n = p_{d_m} \cdot p_d^{n-1} \tag{D.1}$$

As could be expected, increased p_{d_m} results in a reduced completion probability and an increase in the failure probability. It is clear that systems where the probability of detectable fault is large exhibit greatest change in their behaviour due to p_{d_m}; a result which is not entirely surprising since the absolute difference in the probability of detectable fault is necessarily greater for such systems. All in all, the completion and failure probability plots, figures D.1 to D.8, illustrate that increased p_{d_m} does not significantly affect the behaviour of a recovery block system: there is a change in the *value* of the completion and failure probability curves, but their shape is essentially the same as observed previously, figures 6.5 to 6.8.

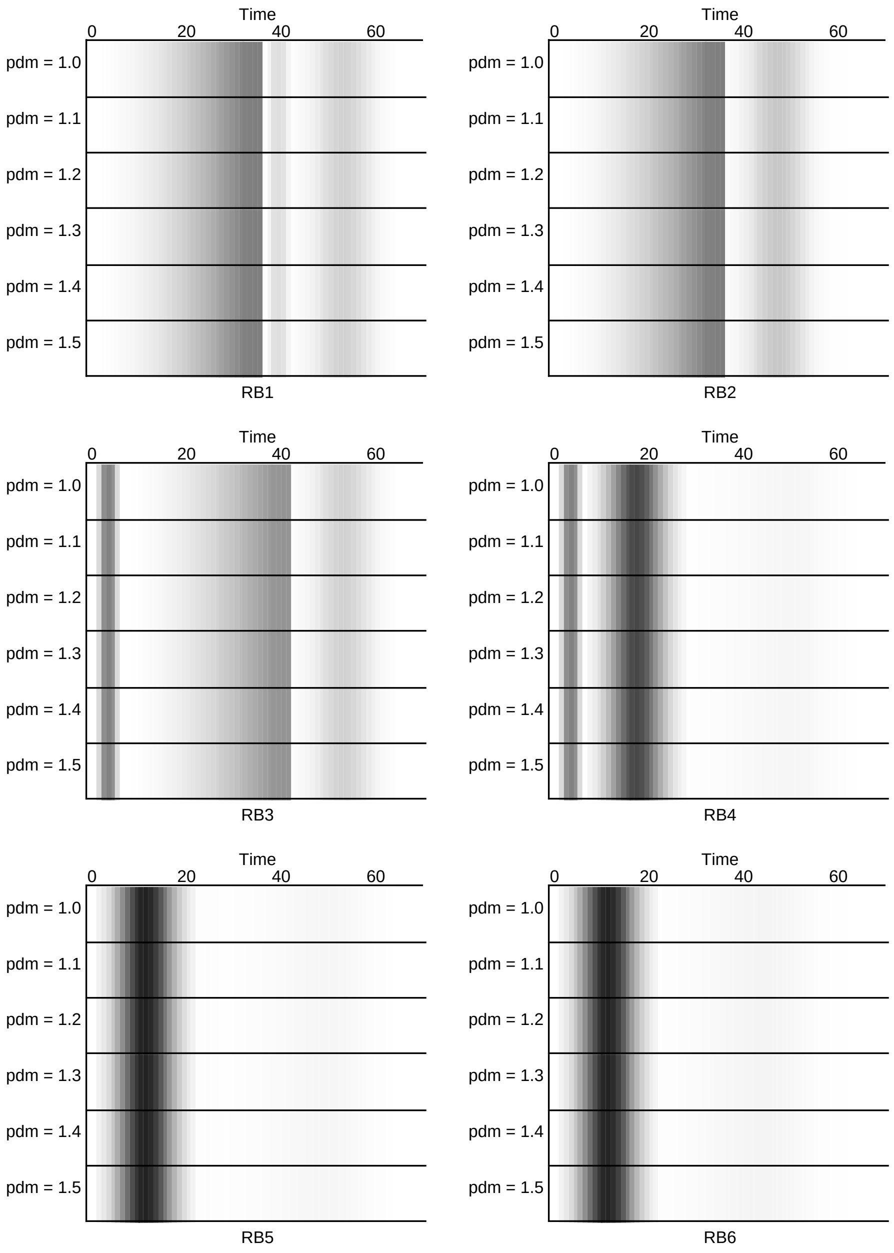

Figure D.1: Instantaneous Completion Probability, $p_d = 0.00050$

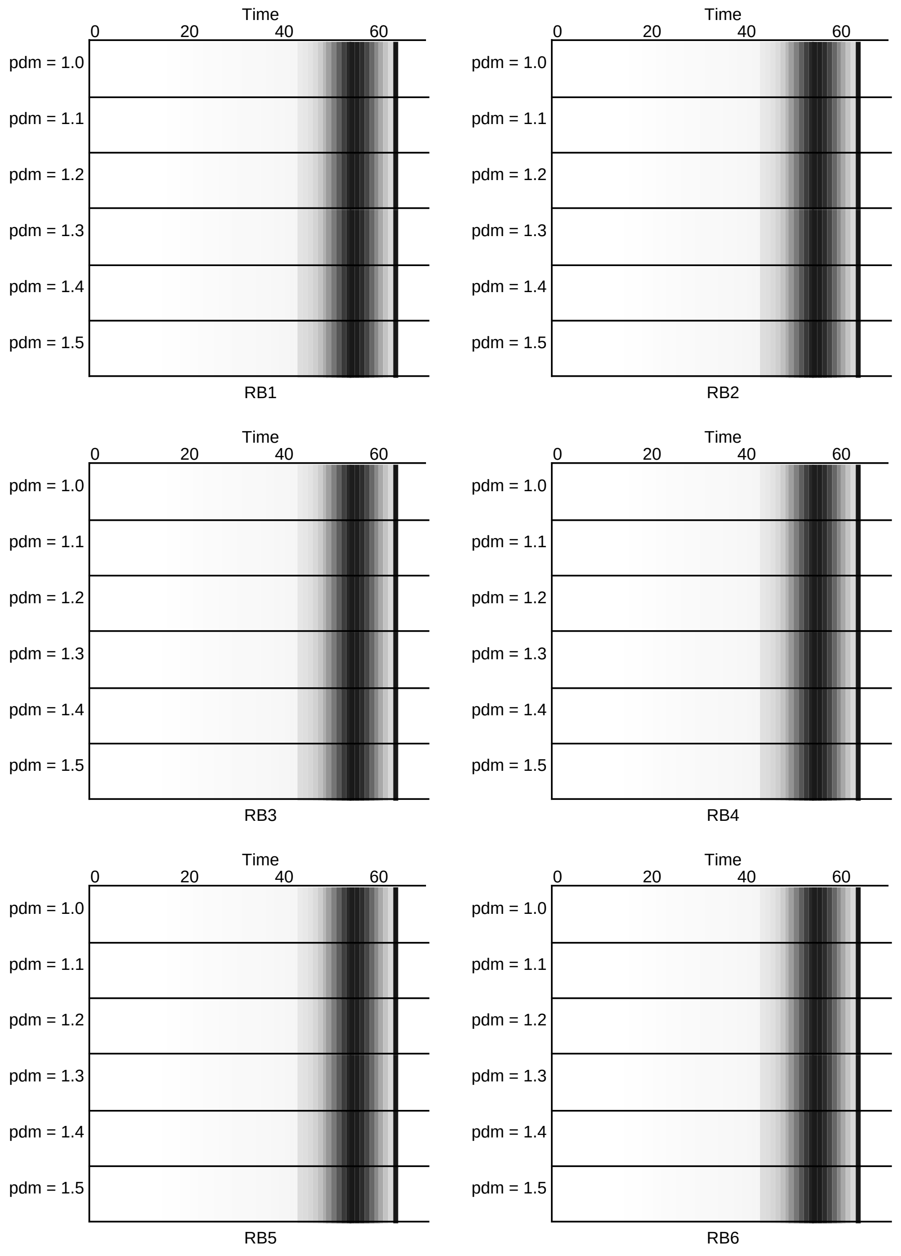

Figure D.2: Instantaneous Failure Probability, $p_d = 0.00050$

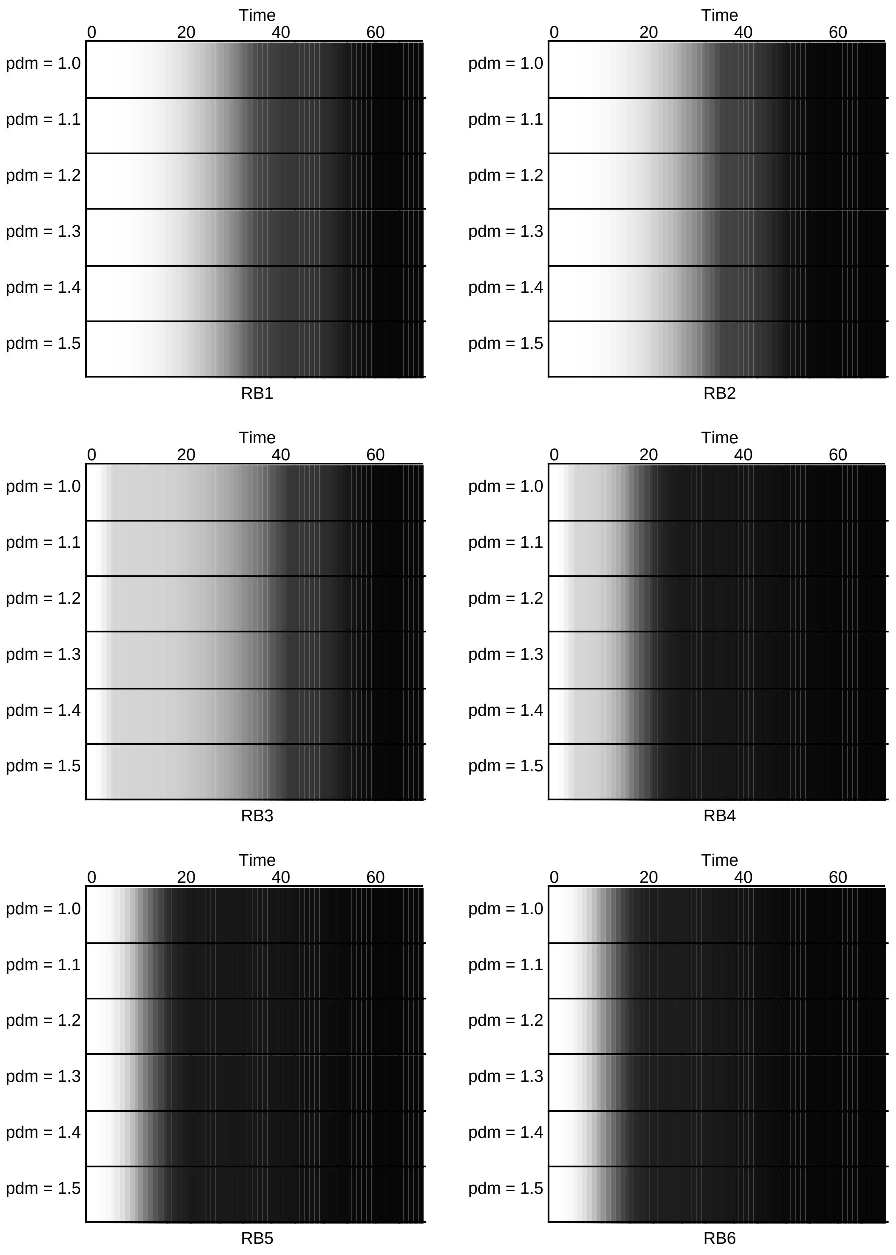

Figure D.3: Cumulative Completion Probability, $p_d = 0.00050$

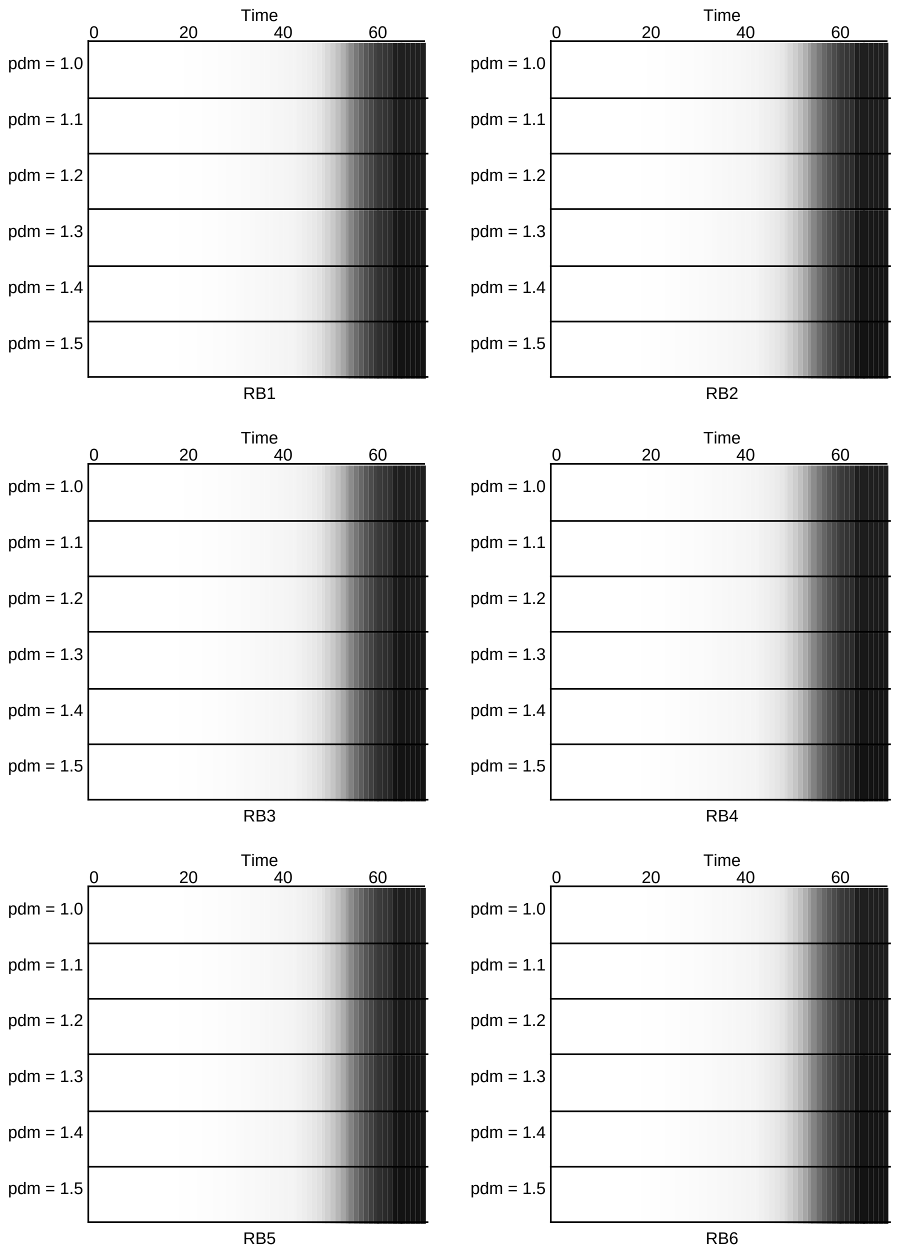

Figure D.4: Cumulative Failure Probability, $p_d = 0.00050$

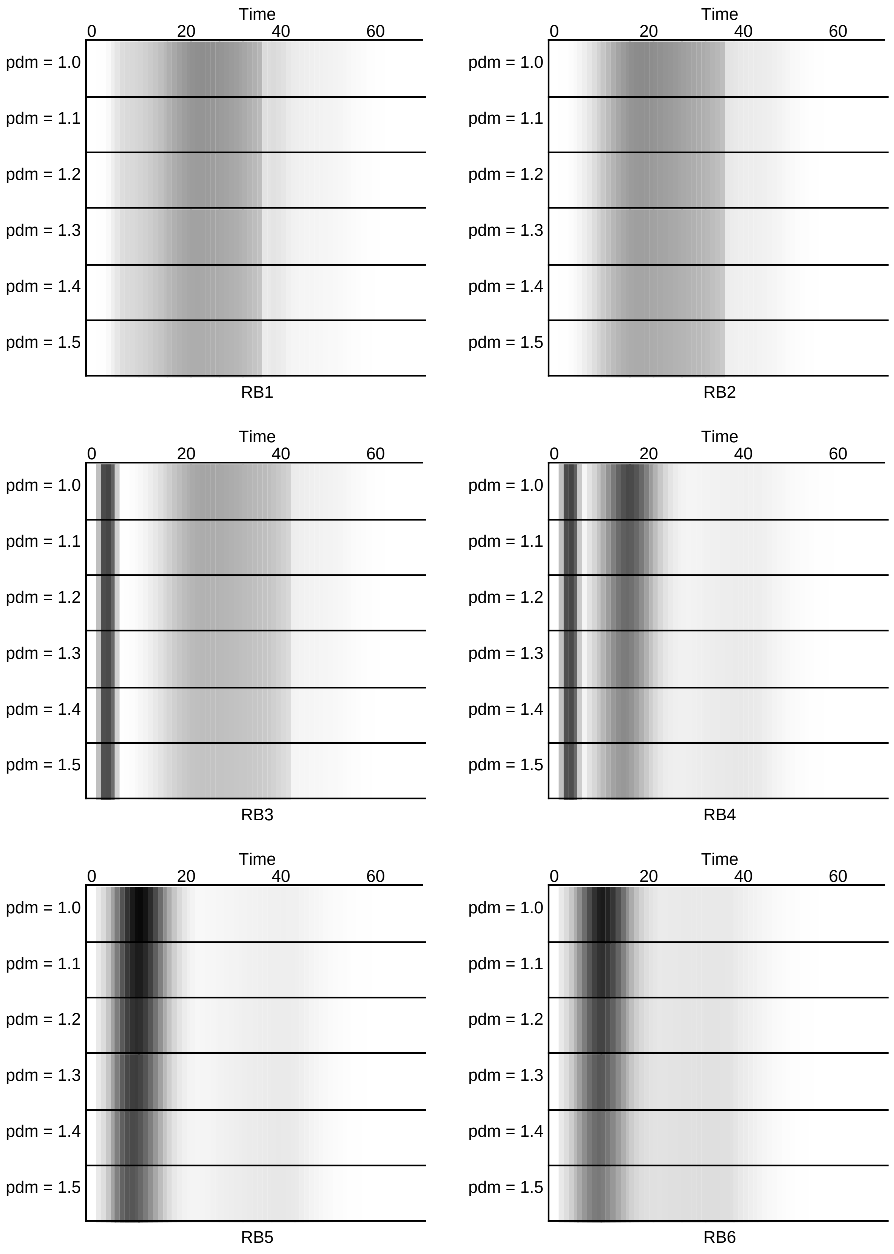

Figure D.5: Instantaneous Completion Probability, $p_d = 0.05000$

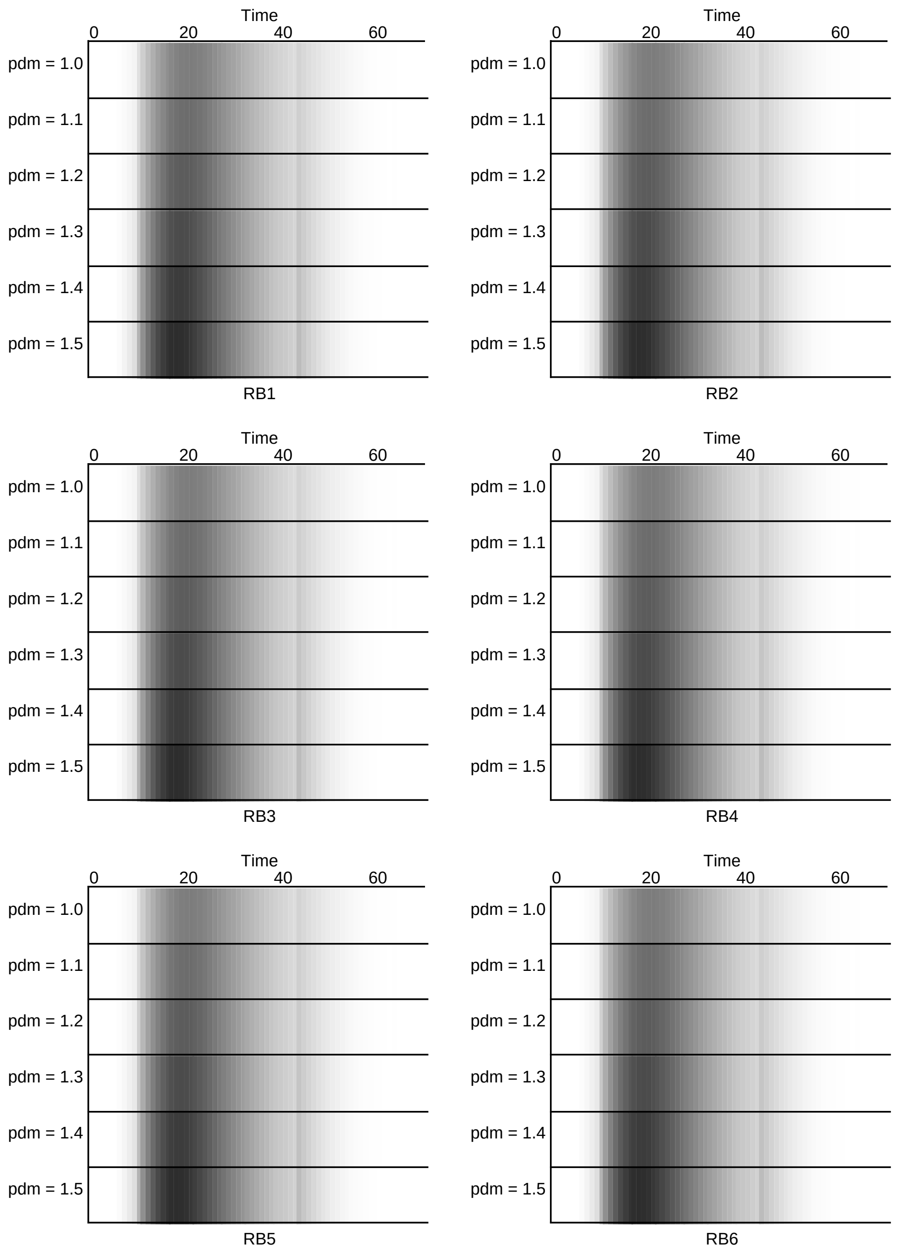

Figure D.6: Instantaneous Failure Probability, $p_d = 0.05000$

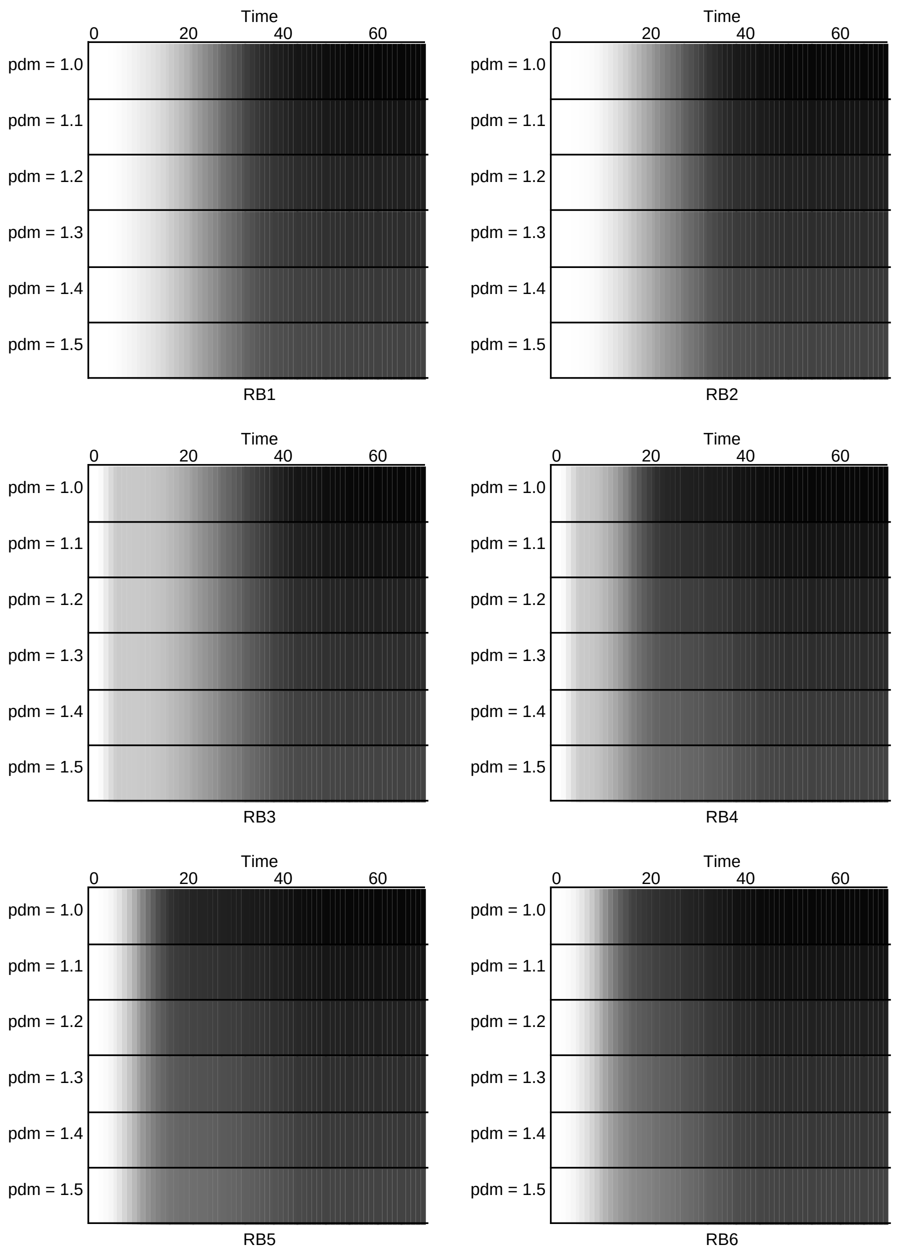

Figure D.7: Cumulative Completion Probability, $p_d = 0.05000$

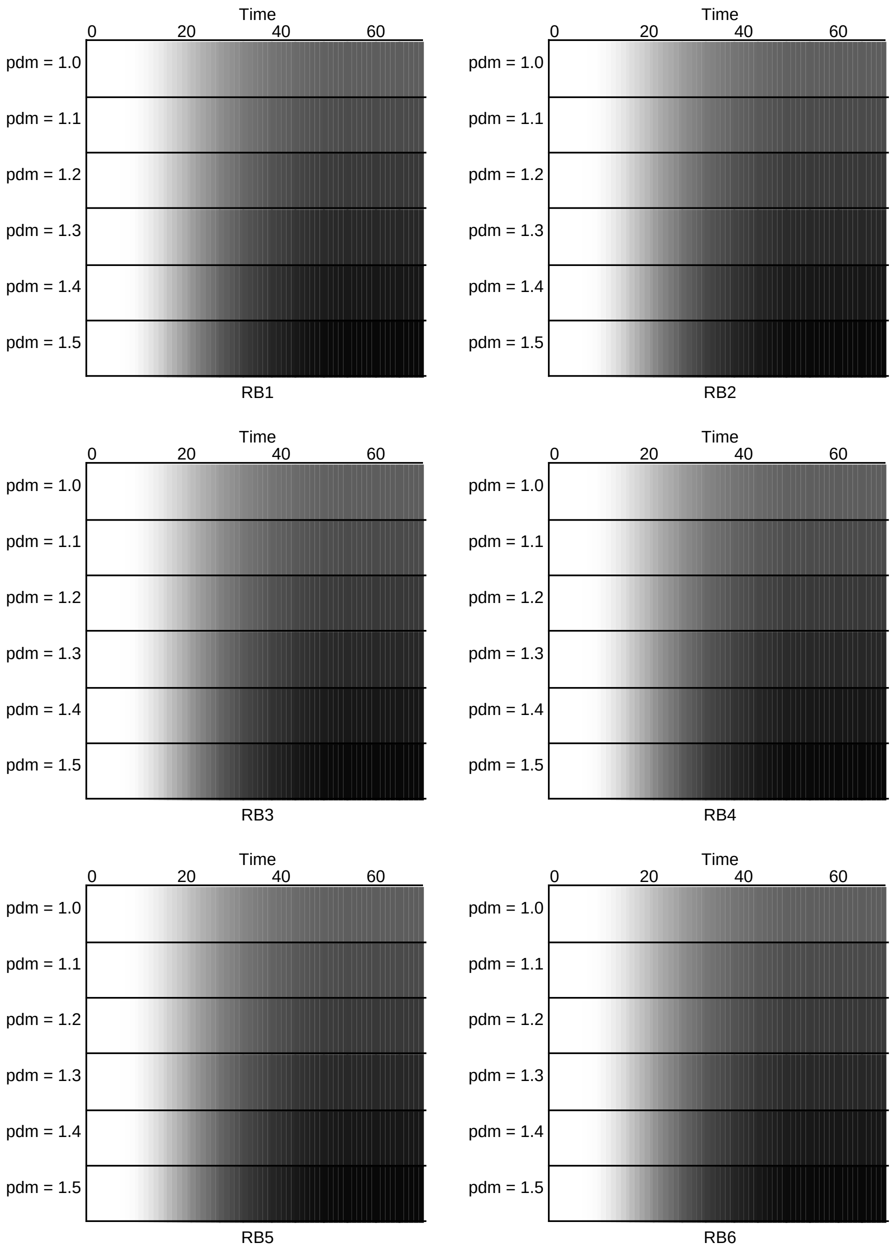

Figure D.8: Cumulative Failure Probability, $p_d = 0.05000$

D.2 Effects of p_f

This section provides a detailed analysis of the recovery block system discussed in section 6.2.3.2. This system uses the same set of alternates, figure 6.3, and p_f and p_r parameters, table 6.1, as that described in section 6.2.2. In addition, p_d was set equal to 0.00500.

The effects of dependent faults in the alternates of the recovery block are here modelled by means of a *failure probability multiplier*. This is a small factor, p_{f_m}, by which the probability of hidden fault occurrence, p_f, is multiplied in each alternate. That is, the value of p_f in alternate n is defined by equation D.2:

$$p_f^n = p_{f_m} \cdot p_f^{n-1} \tag{D.2}$$

The instantaneous completion probability for these systems is shown in figure D.9. The results are clear: increasing the value of p_{f_m} does not significantly affect the shape of the completion profile, but the value of the instantaneous completion probability is modified. In particular, it is seen that some parts of the curve show reduced completion probability, whilst others show an increase. This effect is clearly seen in the cumulative completion probability plots, figure D.11, where it is seen that there is a divergence in the cumulative completion probability plots, dependent on the value of p_{f_m}. It is noted that this effect depends heavily on the ordering of the execution of the alternates: the RB1 system, for example, shows little change in its behaviour; whereas the RB6 system is heavily affected.

The failure probability plots, figures D.10 and D.12 show the expected increase in failure probability as p_{f_m} is increased. It is noted that, once again, all six recovery block systems behave in an identical manner.

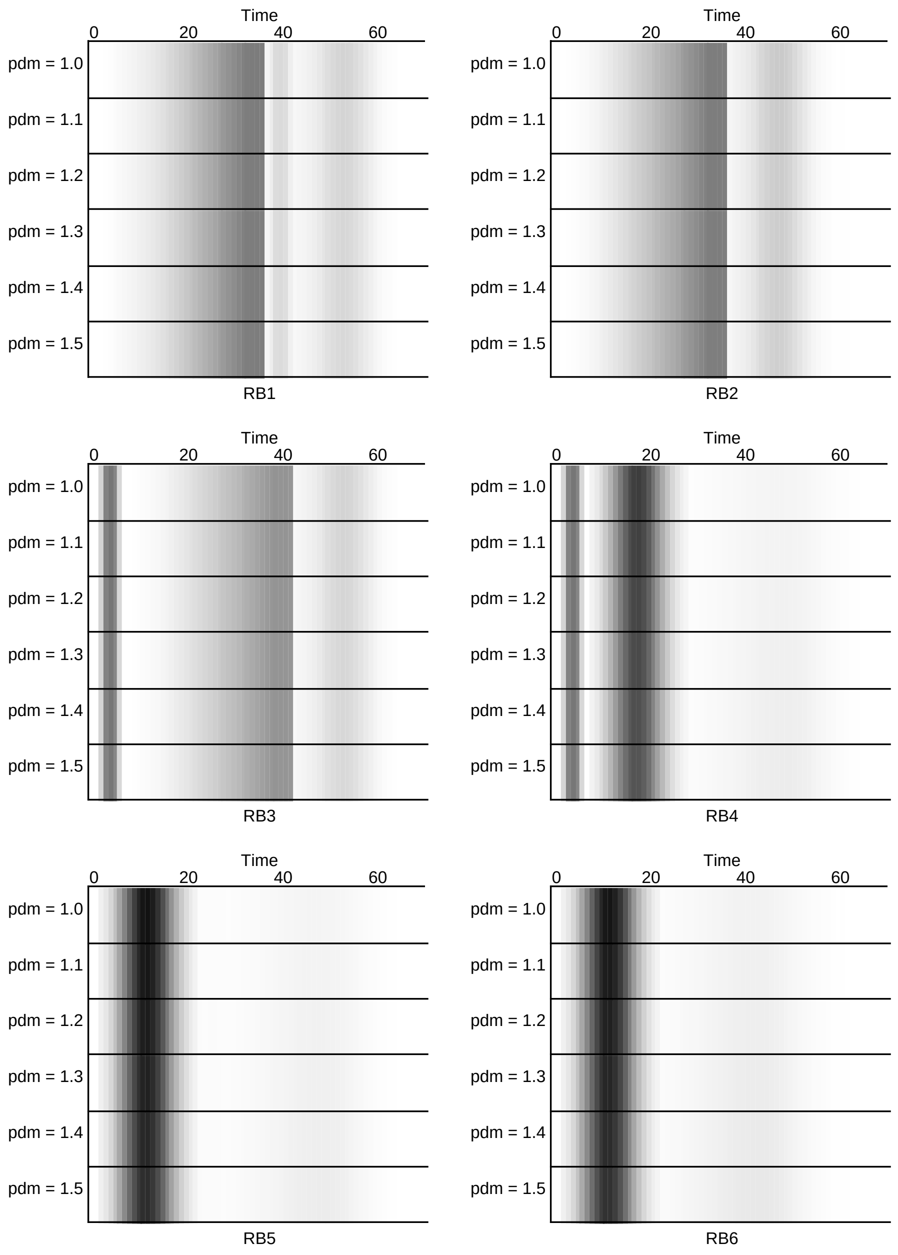

Figure D.9: Instantaneous Completion Probability

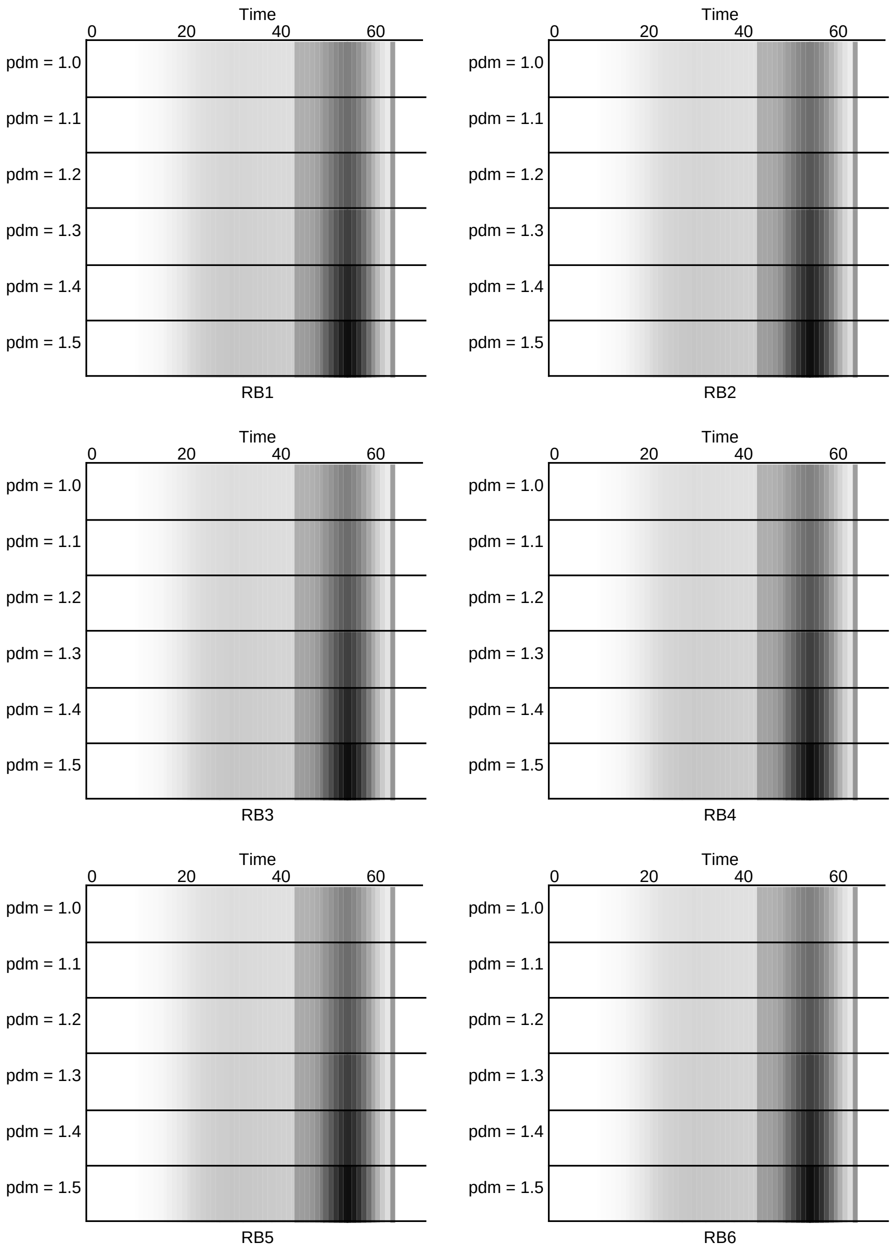

Figure D.10: Instantaneous Failure Probability

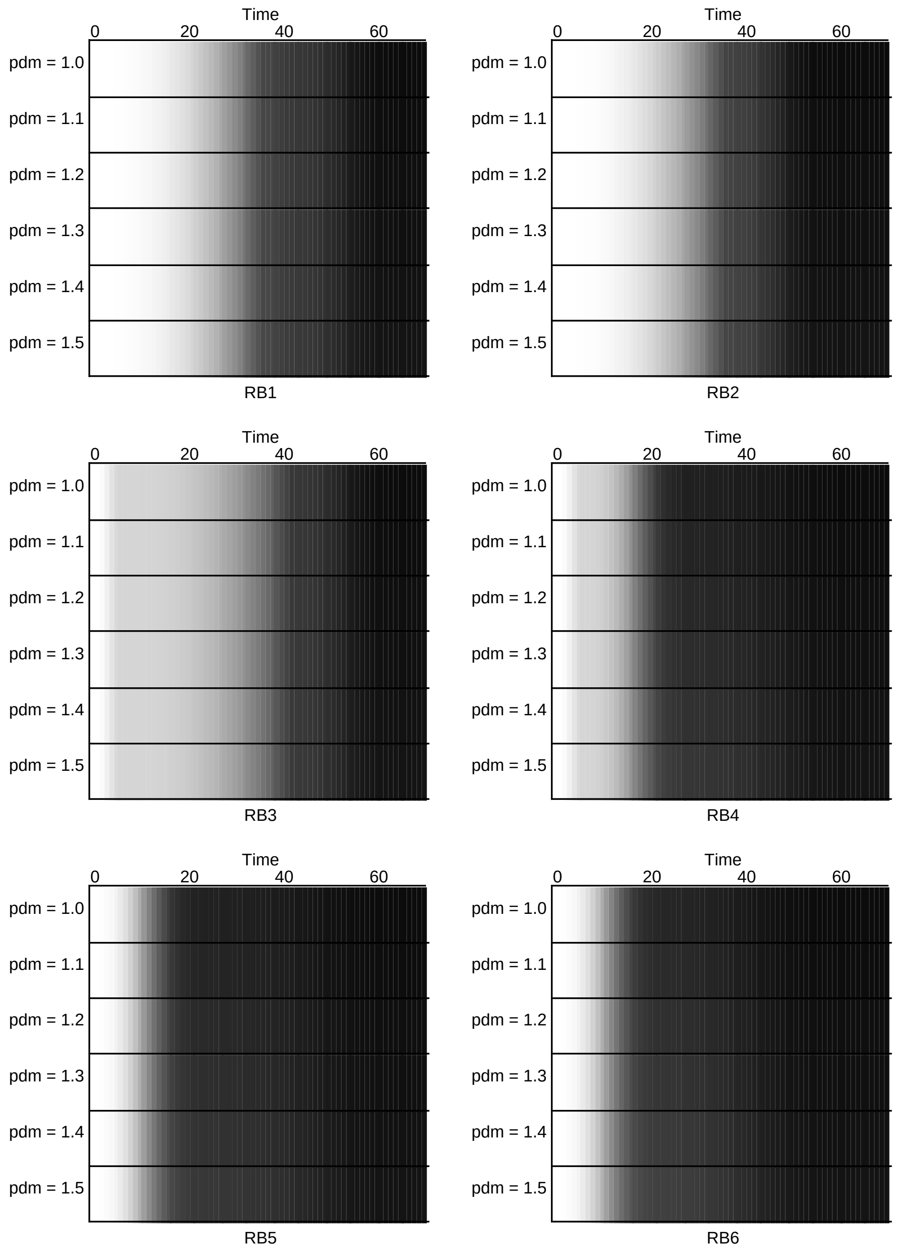

Figure D.11: Cumulative Completion Probability

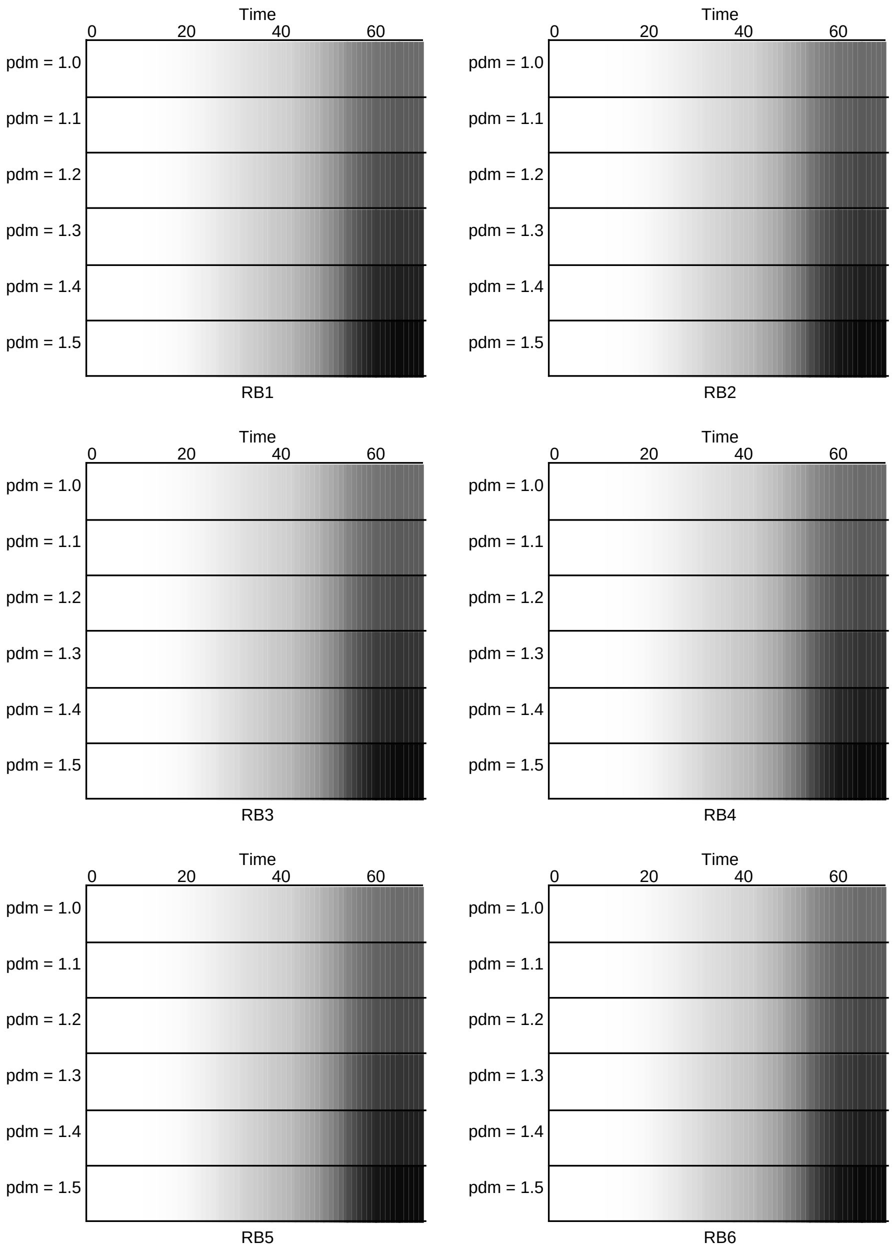

Figure D.12: Cumulative Failure Probability

Bibliography

[1] ACM Committee on Computers and Public Policy. Forum on risks to the public in computers and related systems. Internet mailing list `<risks@csl.sri.com>`. Archived at `http://catless.ncl.ac.uk/Risks/`.

[2] T. Anderson and P. A. Lee. The provision of recoverable interfaces. In *Digest of papers : 9th International symposium on fault-tolerant computing*. IEEE, 1979.

[3] T. Anderson and P. A. Lee. *Fault tolerance — Principles and practice*. Springer-Verlag, 1990.

[4] C. Andre. Synchronized elementary net systems. In G. Rozenberg, editor, *Advances in Petri nets 1989*, volume 424 of *Lecture notes in Computer Science*, pages 51–76. Springer-Verlag, 1989.

[5] J. Arlat, K. Kanoun, and J.-C. Laprie. Dependability modeling and evaluation of software fault-tolerant systems. *IEEE Transactions on computers*, 39(4):504–513, April 1990.

[6] J. Arlat, L. Kanoun, and J.-C. Laprie. Dependability evaluation of software fault-tolerance. In *Proceedings 18th International Symposium on Fault- Tolerant Computing*. IEEE, June 1988.

[7] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Incorporating unbounded algorithms into predictable real-time systems. *Computer Systems Science & Engineering*, 8(2):80–89, April 1993.

[8] A. Avizienis, M. Lyu, and W. Schutz. Multi-version software development: A UCLA/Honeywell joint project for fault-tolerant flight control systems. Technical Report CSD-880034, Dept. Computer Science, University of California, Los Angeles, 1988.

[9] S. Balaji, L. Jenkins, L. M. Patnaik, and P. S. Goel. Workload redistribution for fault-tolerance in a hard real-time distributed computing system. In *19th International Symposium on Fault-Tolerant Computing*, pages 366–373. IEEE, 1989.

[10] M. Balakrishnan and C. S. Raghavendra. An analysis of a reliability model for repairable fault-tolerant systems. *IEEE Transactions on Computers*, 42(3):327–339, March 1993.

[11] Bell laboratories. The ESS No. 1A processor. *Bell Systems Technical Journal*, 56(2), February 1977.

[12] P. A. Bernstein. Sequoia: A fault-tolerant tightly coupled multiprocessor for Transaction Processing. *IEEE Computer*, 21(2):37–45, February 1988.

[13] A. Burns and A. J. Wellings. *Real time systems and their programming languages*. Addison-Wesley, 1990.

[14] J. Campos and M. Silva. Structural techniques for performance bounds of stochastic Petri net models. In G. Rozenberg, editor, *Advances in Petri nets 1992*, volume 609 of *Lecture notes in computer science*, pages 352 – 391. Springer Verlag, 1992.

[15] J. Carlier, Ph. Chretienne, and C. Girault. Modelling scheduling problems with timed Petri nets. In *Advances in Petri nets 1984*, volume 188 of *Lecture notes in computer science*, pages 62–82. Springer Verlag, 1984.

[16] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general purpose distributed computing systems. *IEEE Transactions on software engineering*, 14(2), February 1988.

[17] G. Chiola, M. A. Marsan, G. Balbo, and G. Conte. Generalized stochastic petri nets: A definition at the net level and its implications. *IEEE Transaction on Software Engineering*, 19(2):89–107, February 1993.

[18] K. L. Chung. *Markov Chains with stationary transition probabilities*. Springer-Verlag, second edition, 1967.

[19] G. F. Clement and R. D. Royer. Recovery from faults in the No. 1A processor. In *Digest of papers: 4th Annual International Symposium on Fault Tolerant Computing*, pages 5.2–5.7, January 1974.

[20] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2), February 1991.

[21] A. Csenki. Reliability analysis of recovery blocks with nested clusters of failure points. *IEEE Transactions on Reliability*, 42(1):34–43, March 1993.

[22] R. A. DeMillo, A. J. Offutt, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11, April 1978.

[23] B. Dimitrov, Z. Khalil, N. Kolev, and P. Petrov. On the optimal total processing time using checkpoints. *IEEE Transactions on Software Engineering*, 17(5):436–442, May 1991.

[24] C. I. Dimmer. The Tandem Non-Stop system. In T. Anderson, editor, *Resilient Computer Systems*, pages 178–196. Collins, 1985.

[25] D. E. Eckhardt and L. D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, December 1985.

[26] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on computers*, 41(5), May 1992.

[27] J. R. Elphick. *Fault Tolerance in Rotorcraft Digital Flight Control Systems*. PhD thesis, Department of Electronics, University of York, Heslington, York, YO1 5DD, UK., January 1996.

[28] Engineering Council. Guidelines on risk issues. 10 Maltravers Street, London, WC2R 3ER, UK, February 1993.

[29] European Space Agency. Software reliability modeling study, February 1988. Invitation to tender AO/1-2039/87/NL/IW.

[30] R. Geist, A. J. Offutt, and F. C. Harris. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions of Computers*, 41(5):550–558, May 1992.

[31] J. N. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International conference on very large databases*. IEEE, 1981.

[32] J. N. Gray. Why do computers stop and what can be done about it? In *Proceedings 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, Los Angeles, January 1986.

[33] A. Grnarov, J. Arlat, and A. Avizienis. On the performance of software fault-tolerance strategies. In *Proceedings of the 10th International Symposium on Fault-Tolerant Computing*. IEEE, 1980.

[34] P. J. Haas and G. S. Shedler. Stochastic Petri net representation of discrete event simulations. *IEEE Transaction on Software Engineering*, 15(4):381–393, April 1989.

[35] D. Haban and K. G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. *IEEE Transactions on software engineering*, 16(12), December 1990.

[36] Health and Safety Executive. The tolerability of risk from nuclear power stations, 1992.

[37] A. Hein and K. K. Goswami. Combined performance and dependability evaluation with conjoint simulation. In *Proceedings of the 7th European Simulation Symposium*, pages 365–369, Friedrich-Alexander-Universität Erlangen-Nürnburg, October 1995. Society for Computer Simulation.

[38] B. E. Helvik. Modelling the influence of unreliable software in distributed computer systems. In *Digest of papers : 18th International symposium on fault-tolerant computing*, pages 136–141. IEEE, 1988.

[39] M. P. Herlihy and J. M. Wing. System-level primitives for Fault-Tolerant distributed computing. In *Digest of papers : 16th International symposium on fault-tolerant computing*. IEEE, 1986.

[40] P. G. Hoel. *Elementary Statistics*. John Wiley & Sons, Inc., 3rd edition, 1971.

[41] A. J. Hopkins and T. B. Smith. The architectural elements of a symmetric fault-tolerant multiprocessor. *IEEE Transactions on Computers*, C24(5):498–505, May 1975.

[42] A. Jeffrey. *Mathematics for Engineers and Scientists*. Van Nostrand Reinhold (International), Fourth edition, 1989. ISBN 0 278 00083 5.

[43] Z. Jelinski and P. B. Moranda. Software reliability research. In W. Freiberger, editor, *Statistical Computer Performance Evaluation*, pages 465–484. Academic Press, New York, 1972.

[44] K. H. Kim and H. O. Welch. Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Transactions on computers*, 38(5), May 1989.

[45] K. H. Kim and J. C. Yoon. Approaches to implementation of a repairable distributed recovery block scheme. In *Digest of papers : 18th International symposium on fault-tolerant computing*. IEEE, 1988.

[46] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[47] J. C. Knight, N. G. Leveson, and L. D. St.Jean. A large scale experiment in n-version programming. In *Digest of papers : 15th Annual International Symposium on Fault-Tolerant Computing*, pages 135–139, 1985.

[48] J.-C. Laprie. Dependability evaluation of software systems in operation. *IEEE Transactions on Software Engineering*, SE-10(4):701–714, November 1984.

[49] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Definition and analysis of hardware and software fault-tolerant architectures. *IEEE computer*, July 1990.

[50] J.-C. Laprie and K. Kanoun. X-Ware reliability and availability modeling. *IEEE Transactions of Software Engineering*, 18(2):130–147, February 1992.

[51] J. Ledoux and G. Rubino. A counting model for software reliability analysis. Technical Report 750, Institut de Recherche en Informatique et Systemes Aleatoires (IRISA), Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France, July 1993.

[52] N. G. Leveson. Software safety: Why, what and how. *ACM Computing Surveys*, 18(2):125 – 163, 1986.

[53] B. Littlewood. Stochastic reliability-growth: A model for fault-removal in computer programs and hardware designs. *IEEE Transactions on Reliability*, R-30(4):313–320, October 1981.

[54] B. Littlewood. Software reliability prediction. In T. Anderson, editor, *Resilient Computing Systems*, chapter 8, pages 144–162. Collins, 1985.

[55] B. Littlewood. Limits to evaluation of software dependability. In N. Fenton and B. Littlewood, editors, *Software Reliability and Metrics*, chapter 6, pages 81–110. Elsevier Science Publishers Ltd., 1991.

[56] B. Littlewood and D. Wright. A bayesian model that combines disparate evidence for the quantitative assessment of system dependability. In *Digest of papers : 2nd Conference on the Mathematics of Dependable Systems*, University of York, September 1995. The Institute of Mathematics and it's Applications. Invited paper.

[57] M. A. Marsan. Stochastic Petri nets: an elementary introduction. In G. Rozenberg, editor, *Advances in Petri nets 1989*, volume 424 of *Lecture Notes in Computer Science*, pages 1–29. Springer-Verlag, 1989.

[58] M. A. Marsan, G. Conte, and G. Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions of Computer Systems*, 2(2):93–122, May 1984.

[59] P. Merlin. *A Study of the recoverability of Computing Systems*. PhD thesis, Dept. Information and Computer Science, University of California, Irvine, 1974.

[60] M. K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Transactions on computers*, C-31(9):913–917, September 1982.

[61] M. K. Molloy. Discrete time stochastic petri nets. *IEEE Transactions on software engineering*, SE-11(4):417–423, April 1985.

[62] P. Morrison and E. Morrison. *Charles Babbage and his Calculating Engines*. Dover, New York, 1961.

[63] J. E. B. Moss. *Nested Transactions — An approach to reliable distributed computing*. MIT Press, 1985.

[64] J. D. Musa. Validity of execution-time theory of software reliability. *IEEE Transactions on Reliability*, R-28(3):181–191, August 1979.

[65] J. D. Musa. Software reliability data. Technical report, Bell Telephone Laboratories, January 1980. Report obtainable from DACS, Rome Air Development Centre, Rome, New York.

[66] P. M. Nagel and J. A. Skrivan. Software reliability: Repetitive run experimentation and modeling. Technical Report CR-165836, NASA, Washington, DC, February 1982.

[67] E. Nett, R. Kroger, and J. Kaiser. Implementing a general error recovery mechanism in a distributed operating system. In *Digest of papers : 16th International symposium on fault-tolerant computing*. IEEE, 1986.

[68] E. Nett and R. Schumann. Supporting fault-tolerant distributed computations under real-time requirements. *Computer communications*, 15(4), May 1992.

[69] V. F. Nicola and A. Goyal. Modeling of correlated failures and community error recovery in multiversion software. *IEEE Transactions on Software Engineering*, 16(3):350–359, March 1990.

[70] S. Omohundro and D. Stoutamire. *The Sather 1.0 Specification*. International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley, California 94704, USA, December 1994. Available at `http://www.icsi.berkeley.edu/~sather/`.

[71] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[72] C. Y. Park. Predicting program execution times by analysing static and dynamic program paths. *Real-Time Systems*, 5:31–62, 1993.

[73] C. S. Perkins and A. M. Tyrrell. A new markov model for dependability and temporal evaluation of hard real-time systems. In *Proceedings of the 7th European Simulation Symposium*, pages 388–392, Friedrich-Alexander-Universität Erlangen-Nürnburg, October 1995. Society for Computer Simulation.

[74] C. S. Perkins and A. M. Tyrrell. Reliability models for hard real-time systems. In *Proceedings of the 2nd IMA Conference on the Mathematics of Dependable Systems*, University of York, September 1995.

[75] J. L. Peterson. Petri nets. *Computing Surveys*, 9(3):223–251, September 1977.

[76] J. L. Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, 1981. ISBN 0-13-661983-5.

[77] G. Pucci. A new approach to the modeling of recovery block structures. *IEEE Transactions on software engineering*, 18(2):159–167, February 1992.

[78] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1:220–231, June 1975.

[79] A. Ranganathan and S. Upadhyaya. Performance evaluation of rollback-recovery techniques in computer programs. *IEEE Transactions on Reliability*, 42(2):220–226, June 1993.

[80] A. B. Romanovsky and I. V. Sturtz. Unplanned recovery for non-program object. In *Digest of papers : 12th International Distributed Computing Symposium*, Tampe, U.S.A., 1992.

[81] R. K. Scott, J. W. Gault, and D. F. McAllister. Fault-tolerant software reliability modeling. *IEEE Transactions of Software Engineering*, SE-13(5):582–592, May 1987.

[82] T. Shepard and J. A. M. Gagné. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Transactions on Software engineering*, 17(7), July 1991.

[83] Y.-B. Shieh, D. Ghosal, P. R. Chintamaneni, and S. K. Tripathi. Modeling of hierarchical distributed systems with fault-tolerance. *IEEE Transactions on Software Engineering*, 16(4):444–457, April 1990.

[84] D. P. Siewiorek. Fault tolerance in commercial computers. *IEEE Computer*, pages 26–37, July 1990.

[85] A. Z. Spector, J. J. Bloch, D. S. Daniels, R. P. Draves, D. Dunchamp, J. L. Eppinger, S. G. Menees, and D. S. Thompson. The Camelot project. *Database Engineering*, 9(3):149 – 160, September 1986.

[86] J. A. Stankovic and K. Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, 1990.

[87] J. Tain, P. Lu, and J. Palma. Test-execution-based reliability measurement and modeling for large commercial software. *IEEE Transactions on Software Engineering*, 21(5):405–414, May 1995.

[88] L. Takács. *Stochastic processes*. Methuen, 1962.

[89] D. Taylor and G. Wilson. Stratus. In T. Anderson, editor, *Dependability of Resilient Computers*, chapter 10, pages 222 – 236. BSP, 1989.

[90] L. A. Tomek, J. K. Muppala, and K. S. Trivedi. Modeling correlation in software recovery blocks. *IEEE Transactions on Software Engineering*, 19(11):1071–1086, November 1993.

[91] A. Tripathi and J. Silverman. System-level primitives for fault-tolerant distributed computing. In *Digest of papers : 16th International symposium on fault-tolerant computing*. IEEE, 1986.

[92] D. Wilson. The STRATUS computer system. In T. Anderson, editor, *Resilient Computer Systems*, pages 208–231. Collins, 1985.

[93] J. Xu and D. L. Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Transactions on Software Engineering*, 19(1):70–84, January 1993.