Reliability Models for Hard Real-Time Systems

CS Perkins AM Tyrrell
Department of Electronics, University of York
Heslington, York, YO1 5DD, UK.

Abstract

We present a new reliability model for hard real-time systems. This is an extended Markov model, derived from an analysis of the generic properties of hard real-time systems subject to a simple random-fault model. Our model permits analysis of the run-time behaviour of a system, in order to derive the probability profiles of the system's completion/failure times. The model is applied to the analysis of a simple sequential recovery block system, and illustrative examples based on this system are provided. The paper concludes with a discussion of the application of such accurate completion profile information to the design of embedded software systems.

Keywords: Real-time system, Recovery block, Markov model, completion probability profile.

1 Introduction

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. This paper will describe a new approach to reliability modelling for embedded software systems, with emphasis on the applicability of this technique to hard real-time safety critical systems.

The reliability models which have been developed in the literature may be split into two 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 models [1, 9, 18, 21] are typically based around a Markov-chain or other probabilistic 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 hard 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 hard 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 [4, 7]. 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 [20], there has been little work undertaken to determine the probability distribution of the system's execution time. Much of the research conducted with hard real-time systems has focussed on scheduling problems [2, 8, 16, 22, 24], 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, and it is this problem which is addressed in this paper.

It is therefore noted that: 1) Whilst time-independent reliability models are useful, they do not address a number of important problems which must be resolved before these techniques will be of use in designing hard real-time systems; and 2) 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.

Taken together these points illustrate a problem with current approaches to designing hard real-time systems: It is usual for the timing properties of a hard real-time system to be abstracted away so as to give each process a maximum execution time. Provided such a maximum time can be assigned to each process, it is then possible to devise scheduling algorithms which, given sufficient resources, will ensure that all deadlines are met. These algorithms are, however, pessimistic because 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. The thesis of this paper is that, if the probability distribution of the process' execution times is known, it is possible to design a system which relys on this to attain much improved efficiency, whilst still managing to operate within a tolerable level of risk.

The remainder of this paper is split into the following sections: Section 2 describes the motivation behind this work, and places it in the context of other published literature. Sections 3 and 4 describes the underlying

mathematical model used as a basis for the work, whilst sections 5 and 6 describe the application of this to the modelling of a sequential recovery block structure. Finally section 7 summarises the work.

2 Background

It is desired to model the execution of a system in such a manner that the probability distribution of the execution time can be determined, together with the system reliability. This section will propose a scheme by which this can be accomplished.

A number of experimental studies have been conducted into the failure characteristics of software systems [10, 15]. These studies, together with theoretical results such as those presented in [7, 11, 12, 13, 14] seem to indicate that it is possible to achieve a reasonably 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 is typically 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 1.

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.

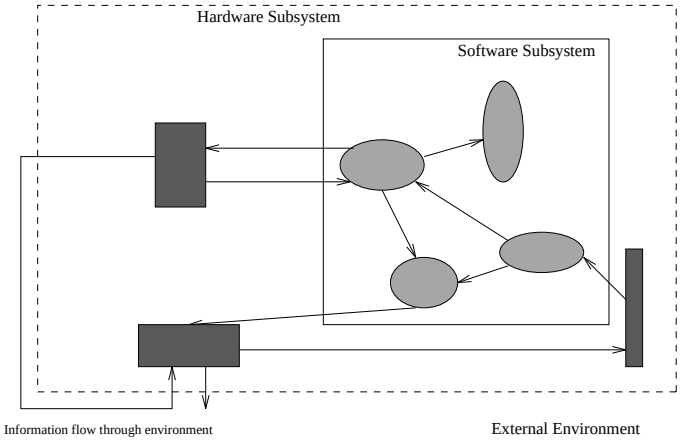

Figure 1. The structure of a typical embedded control system

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 the system will take through its input space [13]. From the above arguments, it seems reasonable to model the 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 2.

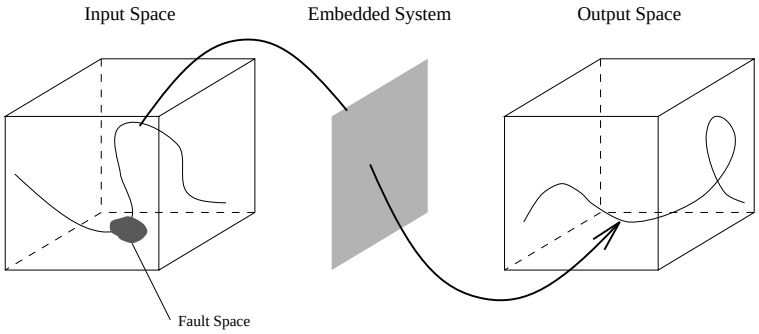

Figure 2. Random walk through the system's input space

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. 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 used in this paper. It will be 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 [12, 13, 14], and certain experimental data [6, 15] has been collected which appears to confirm the validity of this approach. The work conducted by Laprie [11] 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 [6] provides evidence for this claim, as indeed does the work of Musa at Bell Labs [14, 15], and that of the European Space Agency [5], 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.

3 Underlying Mathematical Model

The underlying mathematical model detailed here is a stochastic model, derived primarily from Markov chain theory [23], with modifications to allow for simple process interactions. The underlying network model borrows a number of concepts from Petri net theory [17], not least the notation used. Despite the notational similarities, however, this is primarily a Markov model, not a Petri net system.

The basis of the model comprises 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 these 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.

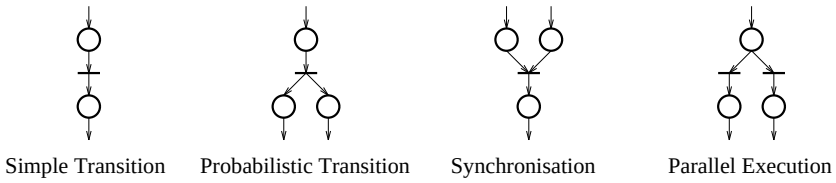

Figure 3. Basic modes of execution

The basic modes of execution of the model are shown in figure 3. 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. 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.

It is the ability to model both probabilistic and concurrent execution with ease which sets this model apart from traditional Markov models.

A formal definition of this model is provided in section 4, whilst details of the application of this model to problems in real-time system design are discussed in sections 5 and 6.

4 Formal definition of the model

4.1 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

- $\Theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$ is a finite set of *places* with $n \geq 2$ representing the system state.
- $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_m\}$ 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 $\Theta \mapsto \Theta$. Furthermore, it is seen that I and O can be regarded as defining arcs connecting the places and transitions of the network. These arcs are weighted, with all arcs having 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 state must be unity.

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

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

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

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

- 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:

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

These definitions provide the basic system structure.

4.2 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 weight of the arc linking places Θ_i and Θ_j.

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:

$$\lambda_k = O(\theta_i) \cap I(\theta_j) \tag{4.14}$$

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

$$\pi^{(t)}_{i,j} = T^{(1)}_{i,j} \prod_{I(\lambda_k)-\theta_i} P_k(t) \tag{4.15}$$

where λ_k is defined in equation 4.14 and $P_k(t)$ is the marking for place θ_k at time t, defined by equation 4.21. 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.

4.3 n-step execution rules

In section 4.1 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 $T_{i,j}^{(n)}$ 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, $T_{i,j}^{(1)}$, 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 4.2, it is possible to derive an expression for the *time-dependent* n-step transition probability, $p_{i,j}^{(n,t)}$, 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

$$p_{i,k}^{(n)} = \sum_j p_{i,j} p_{j,k}^{(n-1)} \tag{4.16}$$

where

$$p_{j,k}^{(0)} = \begin{cases} 0 \text{ if } j \neq k \\ 1 \text{ if } j = k \end{cases} \tag{4.17}$$

It is also 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, $T_{i,j}^{(n)}$, although it is *not* possible to simply add $T_{i,j}^{(n)}$ 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:

$$p_{i,k}^{(n)} = \sum_j p_{i,j} p_{j,k}^{(n-1)} + \sum_{m=2}^{n} \sum_j T_{i,j}^{(m)} p_{j,k}^{(n-m)} \tag{4.18}$$

where $p_{j,k}^{(0)}$ is defined as in equation 4.17. 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, $T_{i,j}^{(m)}$, and indirect, $p_{j,k}^{(n-m)}$, 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 4.15), 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:

$$p_{i,k}^{(n,t)} = \sum_j \pi_{i,j}^{(t-n)} p_{j,k}^{(n-1,t)} + \sum_{m=2}^{n} \sum_j T_{i,j}^{(m)} p_{j,k}^{(n-m,t)} \tag{4.19}$$

where

$$p_{j,k}^{(0,t)} = \begin{cases} 0 \text{ if } j \neq k \\ 1 \text{ if } j = k \end{cases} \tag{4.20}$$

4.4 Markings and system state

The system marking function is defined to be the *absolute probability distribution*, $P_i(t)$ representing the probability that there is at least one token at place θ_i at time t.

The definition of this is based around the equivalent definition for a Markov chain system, modified to allow for time-dependent transition probabilities (as detailed in section 4.2), and to allow for direct n-step transitions, as detailed in section 4.3. This leads to a definition for the absolute probability as follows:

$$P_i(t) = \sum_j P_j(0) p_{j,i}^{(t,t)} \tag{4.21}$$

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

5 Generic Hard Real Time System Model

The mathematical framework described in sections 3 and 4 allows the behaviour of real-time systems to be modelled. From this basic framework a lattice structured model is developed that 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

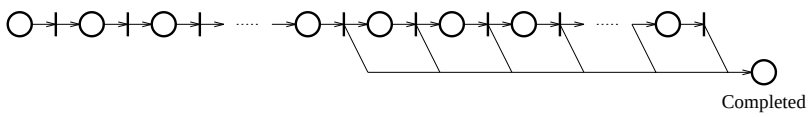

Figure 4. Basic State Chain

functional and temporal properties of a hard real-time system to be represented and is derived from generic properties of hard real-time systems hence being independent of any specific design/implementation technique for such systems. We now discusses the development of this model in some detail.

In order for a system to be classified as *hard real-time* it must obey certain properties. In particular, the system must have well-defined execution time bounds, and the probability of the system exceeding those bounds must be known. 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 4). It is noted that the transition probabilities to the completed state must match the completion profile of the system in the absence of failures. In general therefore it is expected that these transition probabilities will not be uniform.

Such a model is, of course, overly simplistic and must be extended in order to account for the presence of faults within the system. We divide such faults into two classes: (1) Those which cause run-time errors and so are detectable *before* normal system completion; and (2) those faults which do not cause such errors, and so can only be detected by examining the final system state.

The first such 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 *detectable fault* state, with probability determined as using a random fault model (Section 2), this is illustrated in figure 5. Since the system obeys a random-fault model, 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. These parallel states mimic the function of the original state chain, and lead to the *hidden fault* and *failed* states (figure 6). 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. The transitions to/from this parallel set of states have uniform probability, according to the random-fault model.

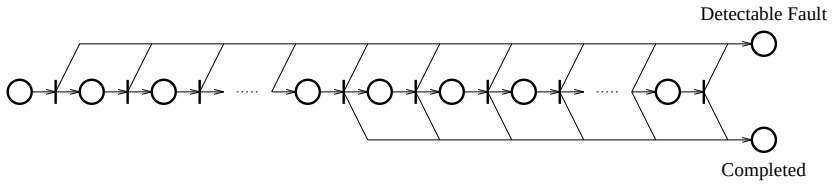

Figure 5. System model with detectable faults

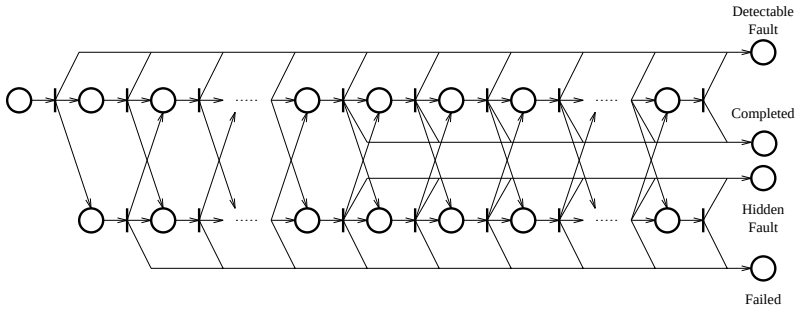

Figure 6. System model with undetectable faults

This then leads to our final model definition, 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, leading to the determination of the marking function (Equation 4.21) for the four final states of this model. It is this marking function which represents the system behaviour with time.

Our model may therefore 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 our 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.

Implicit in the above discussion has been the precise nature of the transition probabilities, $T_{i,j}^{(1)}$, of the lattice model. We divide these into four categories:

Probability of completion, p_c This is the probability that the system completes execution at any given time step. It is independent of the occurrence of faults, and 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.

Probability of detectable fault, p_d This is the probability that the system fails in such a manner that can be detected before the normal completion. It 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.

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 acceptance test. This probability, together with the probability of hidden recovery, defines the transition probabilities on the arcs interconnecting the two main state chains of our model.

Probability of hidden recovery, p_r This is the probability that the system recovers silently from a hidden fault. 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.

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

There is, of course, the question of granularity: At which level is it envisaged that our model will be applied? A typical system will be comprised of a number of fault-tolerant components; perhaps recovery blocks, atomic transactions, or N-version systems with voters. Each of these components will consist of a number of diverse alternates. We feel that our model may best be employed to model the behaviour of these alternates, since these are relatively small systems from which parameters may be readily derived.

Techniques such as those discussed in section 6 may then be employed to extend the model to the component level.

Such a model may then be used as a design aid during schedulability analysis. In particular, it 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.

6 Application to Recovery Block Systems

The recovery block [19] 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 hard 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 hard real-time system, and the model developed in section 5 is applicable. In order to model the full recovery block, an acceptance test model is also required. This must map from the output states of the alternate to the final pass/fail states. A generic acceptance test will be fallible, that is, it will not correctly classify all systems, and will take a finite amount of time. For reasons of simplicity and tractability of the analysis, the test modelled here will, however, be assumed *infallible* [3], and will take unit time. The study of systems with fallible acceptance tests is the subject of current research. This combined alternate and acceptance test model is illustrated in figure 7.

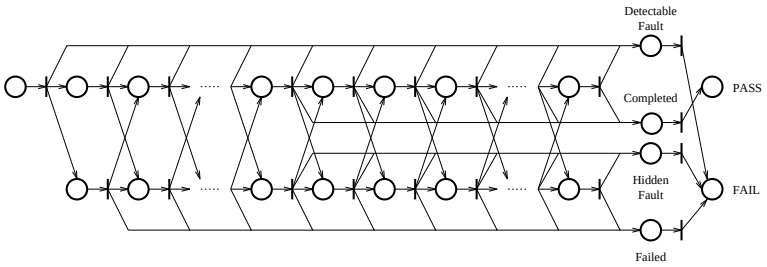

Figure 7. Alternate Model With Acceptance Test

Several such systems may be combined in order to model a complete recovery block. This is illustrated in figure 8 for a recovery block consisting of a primary and two alternates.

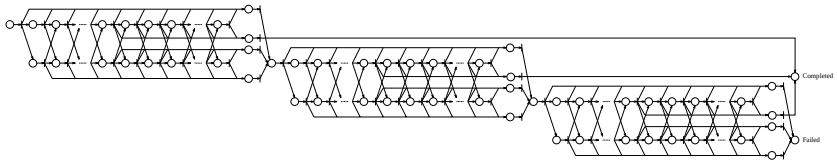

Figure 8. Recovery Block Model

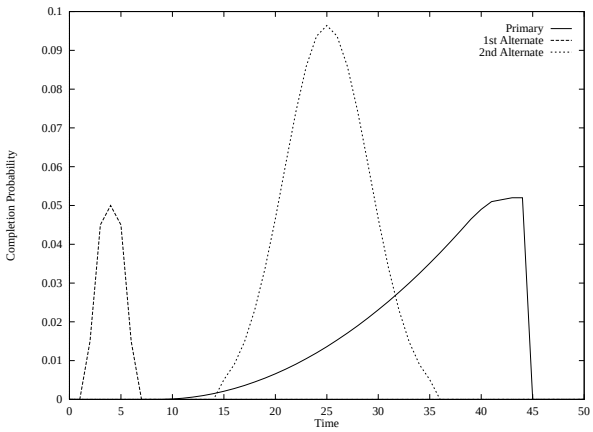

Figure 9. Basic Alternate Completion Profiles

In order to illustrate the applicability of our model, a system such as that in figure 8 has been analysed. For the purpose of this example, the primary and the two alternates were selected as follows (These systems are illustrated in figure 9):

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.

There are three other parameters to the alternate model: Probability of detectable fault, p_d; probability of hidden fault, p_f; and probability of hidden recovery, p_r. In these tests p_f and p_r will be fixed for each alternate, and p_d will be varied. The values chosen for these parameters are shown in table 1.

	Prob. hidden fault, p_f	Prob. hidden recovery, p_r
Primary	0.001	0.001
1st Alternate	0.010	0.010
2nd Alternate	0.010	0.005

Table 1. Alternate Parameters

The results of the analysis in the form of completion/failure profiles for different values of p_c are shown in figures 10 and 11 The completion profile (figure 10) clearly shows the effects of changing the value of p_c, the forward failure rate. For small p_c the system behaves as if the primary and the two alternates are executing sequentially: Indeed this is so, because the majority of failures occuring are due to an alternate exceeding its time bounds, and not due to forward failures. As the failure rate, p_c, increases, the shape of the 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 failure profile (figure 11) shows a related trend. For 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 forward failure rate increases systems are increasingly likely to fail sooner in their life.

Comparing the completion and failure profiles, 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, we see a transition to a mode where alternates rarely complete their execution, and the system failure rate increases dramatically. Such a mode change is confirmed also by a plot of the mean-time-to-failure (figure 12) derived from the failure profile. Such a plot shows a sharp decline around the point where the mode change may be observed on the completion/failure profile plots.

Given this information, and a knowledge of the expected use of the 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. It is believed that in many cases the absolute

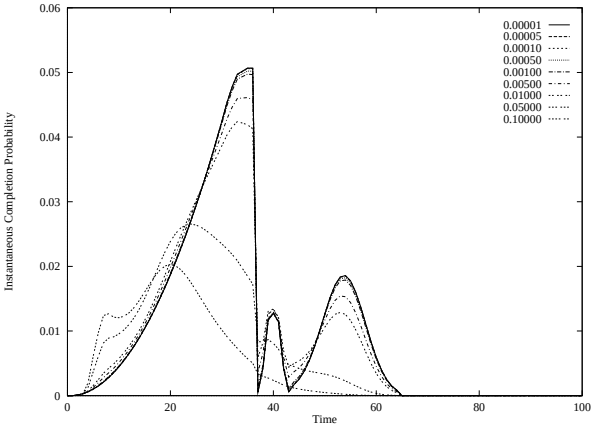

Figure 10. Recovery Block Completion Profile

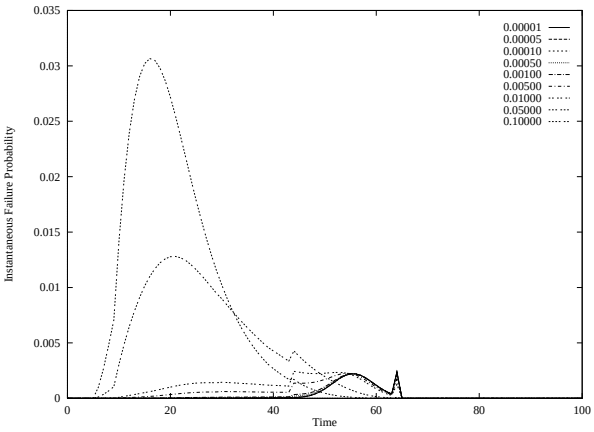

Figure 11. Recovery Block Failure Profile

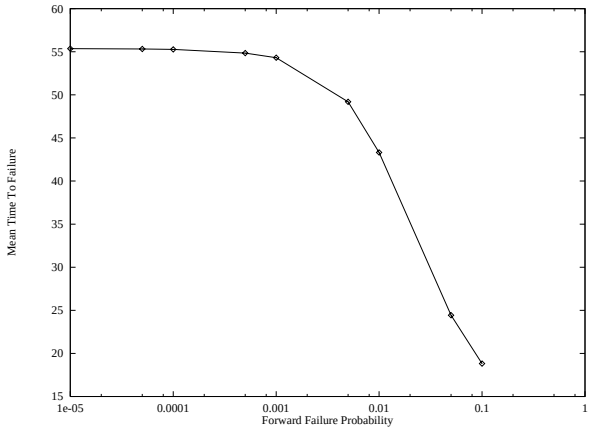

Figure 12. Recovery Block Mean-Time-To-Failure

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 acceptance. Application of this model will provide greater confidence that software can be designed to a specific, *tolerable*, level of risk.

7 Conclusions

In this paper we have introduced a new reliability model for the analysis of hard real-time systems which are subject to faults. The advantage of our technique compared to other published models is its ability to model the completion- and failure-profiles of hard real-time systems as probability distributions with respect to system execution time. The model described is based upon the concept of a random-failure model as applied to an embedded software system, with few parameters. All system parameters should be readily observable from real systems.

The new model has been used to analyse a simple recovery block system. The results illustrate that the model is sufficient for describing such systems. The concept of bounded completion profiles has been introduced, and the use of this technique is system design has been discussed briefly.

Future work will report on the effects of fallible acceptance tests on the system completion profile, and on the application of this model to systems other than the recovery block. In particular, evaluation of concurrent systems requiring synchronisation is a priority.

8 Acknowledgments

This research was supported by the UK Engineering and Physical Sciences Research Council.

Bibliography

1. 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.

2. 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.

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

4. 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.

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

6. 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.

7. 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.

8. 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.

9. 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.

10. 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.

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

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

13. 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.

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

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

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

17. J.L. Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, 1981.

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

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

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

21. 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.

22. 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.

23. L. Takács. *Stochastic processes*. Methuen, 1962.

24. 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.