Informal Proof Analysis Towards Testing Enhancement

Guillaume Lussier, Hélène Waeselynck
LAAS-CNRS
7 Avenue du Colonel Roche
31077 Toulouse Cedex 4 - France
Email: {glussier,waeselyn}@laas.fr

Abstract

This paper aims at verifying properties of generic fault-tolerance algorithms. Our goal is to enhance the testing process with information extracted from the proof of the algorithm, whether this proof is formal or informal: ideally, testing is intended to focus on the weak parts of the proof (e.g., unproved lemmas or doubtful informal evidence). We use the Fault-Tolerant Rate Monotonic Scheduling algorithm as a case study. This algorithm was proven by informal demonstration, but two faults were revealed afterwards. In this paper, we focus on the analysis of the informal proof, which we restructure in a semiformal proof tree based on natural deduction. From this proof tree, we extract several functional cases and use them for testing a prototype of the algorithm. Experimental results show that a flawed informal proof does not necessarily provide relevant information for testing. It remains to investigate whether formal (partial) proofs allow better connection with potential faults.

1 Introduction

This paper investigates the use of proof results for guiding the design of test data. The target application field is the verification of Fault-Tolerance (FT) algorithms.

A number of error processing and fault treatment techniques have been proposed in the literature. In some cases, the underlying algorithms have been formally proved with the support of automated tools (e.g. [18]). But the argument for correctness may also be given as an informal demonstration, in which case there is always the risk of late discovery that the demonstration is flawed and the algorithm does not satisfy its intended requirements. There are examples of such "proofs" of incorrect algorithms in the literature: the case study that will be used in this paper is one of them.

Since FT mechanisms are critical components for building dependable architecture, it is certainly desirable that the argument for correctness is based on strong evidence – but such is not always the case from published results. Moreover, attempts at formal proving may fail in the sense that the proofs remain inconclusive, or partial, due to pending lemmas that could not be discharged.

Testing can be seen as a pragmatic approach to obtain further evidence in case of informal proof results or formal – but partial – ones. Note that, ultimately, testing is always required to validate the implemented FT-architecture. But this is not the issue discussed in this paper. Rather, the concern is the verification of the key properties of the core algorithms. The tested artefact is possibly a prototype of the core algorithm, or a specification that can (in some way) be executed. As an example, the chosen case study is a generic task scheduling algorithm intended to provide support for fault-tolerance. Whether the algorithm correctly addresses the problem it was designed for (under certain assumptions, no task must ever miss its deadline), is preferably verified before developing a particular implementation for a real-time operating system.

Testing an FT algorithm involves sampling over large input spaces, considering inputs related to the functional activity as well as specific inputs related to the faults to be tolerated. Sampling may be guided by coverage criteria based on structural or functional models of the algorithm. But, if testing is intended to complement an existing proof, then ideally it should be focused on the weak parts of this proof. In this way, the effort that was put into the proof development is not completely lost, and testing is directed to the revealing of faults in the algorithm that could not be caught by the flawed proof. Obviously, the test size can be reduced if the required property was formally proved to hold for a subset of the input space. But, even if an informal approach was followed, it might be possible to extract useful information from the proof analysis. For example, a proof by cases might suggest test cases that would be potentially significant to the correctness of the algorithm. Identification of the most complex and less convincing parts of the informal proof might suggest input subspaces to be sampled more stringently than others. However, how these general ideas

can be put into practice, and whether they do allow the test efficiency to be improved, remains to be investigated. This paper reports on work in that direction.

Section 2 presents related work investigating the coupling of different verification techniques, emphasis being put on testing and proving. Section 3 introduces the Fault-Tolerant Rate Monotonic Scheduling (FT-RMS) algorithm used to exemplify our work. Section 4 reports on previous results establishing the presence of (at least) two faults in the algorithm. Section 5 and 6 are the core of the paper. Section 5 contains a detailed discussion of the informal demonstration of the algorithm, which we restructure as a semi-formal proof tree based on natural deduction. In Section 6, test experiments are conducted in order to 1) assess the difficulty of revealing the two faults using a blind test strategy and 2) assess the fault revealing power of test data in relation to coverage of functional cases extracted from the proof analysis, as well as in relation to structural coverage of the algorithm. Section 7 concludes and gives directions for further investigation.

2 Related Work

Fruitful cooperations of verification techniques have already been developed, particularly in the field of protocol validation. One example is the integration of model-checking techniques into the PVS prover. The principle of this approach [20, 22] is to use a tight interaction between theorem proving and abstraction generation for model-checking. Every step consists of applying one of the techniques and makes constructive use of information obtained from previous steps. The iterative process, when it terminates, yields a counterexample indicating how the property is violated or a proof that the property is satisfied. Another example is the benefit that protocol testing has gained from advances made in the model-checking technology. Methods and tools have been developed to automate the generation of test suites, either by adapting graph algorithms already used by model-checkers [7], or by using the ability of model-checkers to construct counterexamples [1, 8].

Practical ways to combine testing and theorem proving have been less explored in the literature. Yet the idea has been expressed since the 70s: *"A judicious combination of direct program proving and empirical judgment can reduce size of a complete test."* [12].

From the formal method community, an original work is Hayashi's in [13] who proposes to use the *Curry-Howard isomorphism* to debug formal proofs, including partial ones. This isomorphism establishes a correspondence between programs and proofs in constructive logics, hence allowing to extract programs from the proofs developed. It states that in such logics, proofs and programs are equivalent according to the rules of this isomorphism. The method developed by Hayashi, called Proof Animation, consists in "executing" proofs as functional programs, using test cases. It is intended to facilitate proof construction and a tool, Proof-Works, was built to help finding "bugs" (proof flaws) in proofs under development and even in fully checked proofs (formalization flaws). But the use of the Curry-Howard isomorphism restricts the application of the method to constructive logics which are unfamiliar to most users.

Regarding the works originating from the testing community, it has primarily been concerned with the generation of test data from formal specification, for conformance testing. It has been less concerned with developing tight interaction between testing and proving, in the framework of mixed verification strategies. Geller tackled the problem in [9] for the verification of program assertions. He used an inductive approach, by testing specific values from the input domain and then by generalyzing the results to *equivalence classes* using a mathematical proof. (An equivalence class is a set of inputs equivalent in their capability of stimulating the system under test.) But in an inductive approach, the most difficult problem is the generalization proof and not the proof of the base cases: hence testing does not significantly help proving in this approach. In the nearby field of partition analysis, Richardson and Clarke propose in [19] a method combining functional and structural partitioning into equivalence classes. The aim is to guide the selection of test data as well as to formally verify consistency between the two partitions obtained. Another approach defined by Cukic [2] uses slicing techniques and partial proofs to reduce the size of the test input domain, for reliability assessment. More closely related to our purpose, the work of Sinha and Suri [23, 24] investigates an approach using formal verification procedures to guide fault injection based validation. They built a model of the FT-RMS algorithm in the PVS specification language and queried this model to study the correctness of the arguments presented in the informal proof of the FT-RMS. The queries were focused on specific task sets, allowing the identification of two classes of test scenarios leading the FT-RMS to fail. A more detailed presentation of these results will be found in Section 4, and we will use the same case study as a first step to our investigation.

3 The FT-RMS algorithm

The FT-RMS algorithm has been implemented on two real-time operating systems [3, 6] : RT-Mach, an academic system, and DEOS (Digital Engineering Operating System), a commercial system for avionics applications. Quoting from [5], "real-time scheduling problems are an ideal application area for formal methods since they are subtle and complex, [and] must be certified to the highest degrees of assurance for supporting critical applications". The first

definition of FT-RMS in [10] contained (at least) two faults, one of them being fixed in the revised version of the algorithm published in [11]. As mentioned in the previous section, the algorithm has also been studied by others [23], yielding the second fault to be revealed. It is worth noting that the RT-Mach and DEOS implementations correspond to variants of the algorithm, and may indeed be correct. In this paper, the focus is on the incorrect versions known of the algorithm presented in [10, 11].

In order to distinguish between the faults the FT-algorithm is intented to deal with, and the design faults affecting the algorithm itself, we will use the term "flaw" to denote the latter category of faults in the rest of this paper. Before explaining the principles of the FT-RMS, we briefly introduce the Rate Monotonic Scheduling (RMS) policy on which it is based.

3.1 The RMS Algorithm

RMS is a fixed-priority scheduling algorithm for sets of independent, periodic tasks. Each task τ_i is characterized by its computation time C_i and period T_i. The k_{th} instance of task τ_i, activated at time $t_{0i}+(k-1)T_i$, must be completed before the next request for τ_i occurs at time $t_{0i}+kT_i$. The *rate monotonic* scheme assigns priorities to tasks according to their request rate, that is, tasks with smaller periods will have higher priorities. The ready task with the highest priority is always chosen for execution; execution of a task is pre-empted whenever there is a request for a task that is of higher priority. Since priorities are assigned to tasks once and for all, the order of execution of tasks may be determined off-line (static scheduling). Fixed-priority scheduling schemes have been extensively studied. We recall herebelow some theoretical schedulability results.

Liu and Layland [17] established a sufficient condition for a set of n tasks to be schedulable using RMS:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq U_{LL} = n(2^{1/n} - 1) \tag{1}$$

U is the *utilization factor*, that is, the fraction of processor time spent in the execution of the task set. The bound on U is obtained by worst-case analysis. Liu and Layland introduce the notion of *critical instant* at which all tasks become ready to start computation, so that the task requests will have the largest response time. Then, they focus on tasks sets that *fully utilize* the processor: at the critical instant, scheduling is feasible for those sets, but any increase in the computation time of one of the tasks makes it unfeasible. Those task sets have the general form:

$$\begin{cases} C_i = T_{i+1} - T_i & \forall\, i.\ 1 \leq i \leq n-1 \\ C_n = 2T_1 - T_n \end{cases} \tag{2}$$

where the n tasks are ordered according to their priority ($T_1 \leq T_2 \leq \ldots \leq T_n$). It is demonstrated that the minimum of the utilization factors, over all the full utilization sets, is $n(2^{1/n}-1)$. As a result, any task set having utilization factor below this value is schedulable.

Relation 1 yields a conservative bound. An exact analysis technique has been proposed by subsequent work. It consists in computing the worst-case response time R_i of each τ_i, and in verifying that $R_i \leq T_i$. Let $\{\tau_1, \ldots, \tau_{i-1}\}$ be the set of tasks having a priority higher than τ_i. Then, R_i is the smallest solution to the following fixpoint equation (see e.g. [14]):

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j \tag{3}$$

3.2 Two versions of the FT-RMS Algorithm

The FT-RMS algorithm extends RMS to include fault tolerance capabilities. Faults are assumed to be transient: only one task instance is affected by each fault and recovery is carried out by re-execution. Error detection takes place at the end of every task execution. Its cost is assumed to be incorporated in the tasks C_i. Additional hypotheses concern the amount of time between any two faults, but we will ignore them since the incorrect behavior of the algorithm can be revealed by considering a single transient fault.
The principle of the algorithm is to use RMS when there is no fault. After an error is detected, the affected task is re-executed using a certain recovery scheme. Of course, there must be enough time redundancy so that neither the re-executed task nor the other tasks, miss their deadlines. This is handled by provision of backup time (or *slack*) reserved for re-execution, and by the *recovery scheme* describing how the slack is used for dynamic re-execution. The provision of slack is identical in the two versions of the algorithms published in [10, 11] , but the recovery scheme is not the same.

The provision of slack imposes a condition on the target set of tasks. This set must be schedulable using an RMS scheme with slack inserted, called the IBRMS scheme (for Inserted Backup RMS). According to IBRMS, the amount of slack inserted over an interval of time is proportional to the length of that interval. The constant ratio U_B is equal to the maximum of the tasks utilization factors, i.e. $U_B = max(C_i/T_i)$. The IBRMS scheme consists in inserting one backup slot between every consecutive period boundaries, where a period boundary corresponds to the arrival of at least one task request. Given two consecutive period boundaries which are L time units apart, the length of the corresponding backup slot is $U_B \times L$. As an example, Figure 1 shows the IBRMS schedule for a given set of 3 tasks. Let us recall that tasks with smaller periods have

τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	16	60	0.276
τ_2	23	73	0.315
τ_3	2	116	0.017
			$U_B = 0.315$

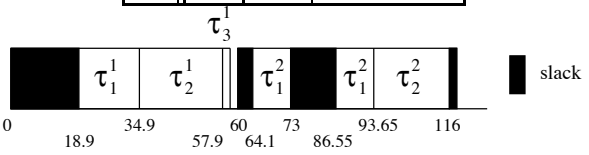

Figure 1. IBRMS schedule

higher priorities, hence τ_1 is assigned the highest priority and τ_3 the lowest. In the construction of an IBRMS schedule, time $t = 0$ is usually assumed to be a critical instant.

Now let us assume that a set of tasks satisfies the IBRMS feasibility condition. During actual task execution, a normal RMS policy is used as long as there is no fault. Hence, the slack can be considered as being swapped with the execution of tasks, and shifted later in time. When an error is detected, slack is reclaimed according to the recovery scheme.

The original recovery scheme in [10] is the simplest. The faulty task is re-executed at its own priority. However, as exemplified by Figure 2.a, this scheme is flawed: higher priority tasks may prevent the faulty task from obtaining the reserved slack within its deadline. This has led a new recovery scheme to be proposed in [11]. In this scheme, re-execution of the faulty task τ_r cannot be pre-empted by higher priority tasks having a deadline strictly greater than the one of τ_r (see Figure 2.b). Unfortunately, as will be explained in Section 4, this scheme was also found incorrect.

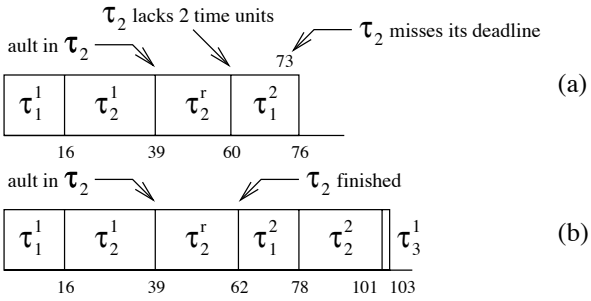

Figure 2. FTRMS schedule, version 1 and 2 (same task set as in Fig. 1)

Whatever the recovery scheme, the informal proof of the algorithm is based on the fact that the set of tasks satisfies the IBRMS feasibility condition. However, the authors also study a more stringent condition related to Liu and Layland's bound on the utilization factor. The proposed bound for FT-RMS is:

$$U_{FT-RMS} = U_{LL}(1 - U_B) \tag{4}$$

This U_{FTRMS} bound was taken into consideration by Sinha and Suri who exposed a flaw in the second recovery scheme.

4 Previous results on FT-RMS verification

As pointed out by [4], current model-checking technology is unable to support complete verification of real-time schedulers. Theorem proving is a valid alternative but requires a lot of effort and may also fail to produce a complete proof. As regards testing, the size of the input space raises the issue of the degree of coverage that can practically be achieved. Hence, FT-RMS was identified by Sinha and Suri as a relevant example to investigate the use of formal approaches to guide the design of testing. Moreover, since the first version of the algorithm was known to be incorrect, they wanted to study the ability of their approach to reveal the corresponding flaw.

The proposed approach [23] analyzes the algorithm behavior using symbolic execution and query processing. This is performed with the aid of PVS. The PVS model of the first version of FT-RMS [24] is not intended to reproduce the concrete algorithm description. For example, the notion of schedule is not captured in the model. Rather, the authors propose an interpretation of the algorithm using equations similar to Equation (3) in Section 3.1. It allows them to simulate the behavior of the algorithm by calculating response times. A first analysis of the model behavior showed that the FT-RMS provision of slack is flawed: there is a discrepancy between the amount of slack that is claimed to be available (according to IBRMS) and the amount of slack that is actually obtained for dynamic re-execution. Based on this result, the authors tuned their verification process by choosing a specific case aimed at minimizing the natural slack present in the schedule. They chose the *full utilization* criterion (see Section 3.1). Strictly speaking, there is no proof that this criterion, originally introduced in the framework of static scheduling schemes, constitutes a worst case for FT-RMS. This is a debatable issue for the second version of the algorithm which has a dynamic priority adjustment rule upon faults (see [16] for a discussion of worst case scenarios for such scheduling schemes). Nevertheless, the focus on full utilization task sets turned out to be effective to reveal a residual flaw.

The considered task sets are defined as follows, where Δ is a positive number as small as desired:

$$\begin{cases} C_i = T_{i+1} - T_i & \forall\, i.\, 1 \le i \le n-1, \\ C_n = (2T_1 - T_n)/2 + \Delta & \\ & with \sum_i U_i \le U_{FT-RMS} \end{cases} \tag{5}$$

Compared to Liu and Layland's original full utilization task sets (Relation 2), the computation time of one task, namely

the last one, is divided by two. This ensures that the task sets fully utilize the processor if the re-executed task is τ_n. But this also reduces the generality of (5) for studying the behavior of FT-RMS, as τ_n is singularized. The small value Δ added to C_n will prevent the successful re-execution of this task in the first version of the algorithm, in spite of the fact that the IBRMS feasibility condition holds true (which is granted by the U_{FT-RMS} bound).

The authors exhibited a set of four tasks satisfying the above numerical constraints, and entered it into the PVS model. They confirmed that the task set leads the first version of FT-RMS to fail if the fourth task is the faulty task to be re-executed. But they also observed a failure when the faulty task is the third one, and this flaw is not fixed in the second version of the algorithm, as shown in Figure 3. As a result, the approach ends up with two classes of test scenarios, both involving task sets that satisfy the above constraints (5). The first one consists in the lowest priority task τ_n being faulty. In the second case, the faulty task is τ_{n-1}.

τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	0.5	4.0	0.125
τ_2	0.5	4.5	0.111
τ_3	1.0	5.0	0.2
τ_4	1.01	6.0	0.1683
			$\Sigma_U = 0.604$

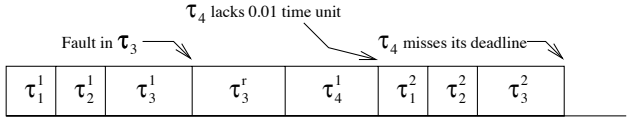

Figure 3. Failure of FT-RMS version 2

These classes of scenarios are intended to be equivalence classes, that is, any member of the class has the ability to reveal the corresponding flaw (respectively the already known one, and the residual one). Unfortunately, we found that this does not hold for the second class of scenarios. A task set will be schedulable under both versions of the algorithm if $C_{n-1} \leq C_n - 2\Delta$, as exemplified by Figure 4. The task set fulfils all the required conditions for the second class of scenarios, but no failure is observed.

τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	0.5	4.0	0.125
τ_2	0.5	4.5	0.111
τ_3	0.5	5.0	0.1
τ_4	1.26	5.5	0.23
			$\Sigma_U = 0.566$

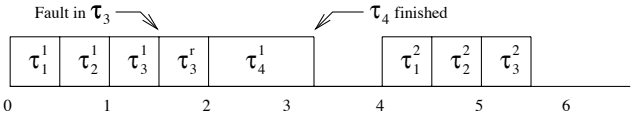

Figure 4. Faultless full utilization task set

The results of [23] were biased by the particular values chosen to instantiate the full-utilization task sets. Different values could have resulted in the second fault to be missed. The example in Figure 4 clearly shows that the processor is not fully utilized when τ_n is not the faulty task.

5 Informal proof analysis

We propose to study the FT-RMS proof itself to gain further information for the design of testing. Note that the previous PVS model [24] cannot be used to conduct a formal proof of the FT-RMS. We would need an operational model of FT-RMS (e.g. a state-machine model capturing the algorithm), lemmas establishing an explicit relation between this model and the response time calculation proposed in [24], as well as mathematical theories for supporting scheduling analysis (e.g., support for reasoning about the fixpoint equations). An example that would better meet these needs is the PVS development presented in [4], which allowed another scheduling algorithm to be formalized and proved (the Priority Ceiling Protocol). It is our intention to study whether this development can be reused for FT-RMS, but we first focus on the informal proof given in [11]. It corresponds to the second version of the algorithm.

5.1 Description of the approach

We analyze the informal proof of the FT-RMS with the double aim of (i) trying to identify the weak parts of the proof, and (ii) extracting functional cases that would be relevant for testing. We use Gentzen's *natural deduction* in sequent style to restructure the informal discourse and analyze the resulting proof tree. Natural deduction is convenient to represent usual reasoning. A *sequent* is written in the form $\Gamma \vdash P$, where Γ is a list of hypotheses, and P is a conjecture to be proved under these hypotheses. An intuitive interpretation is that the conjunction of the hypotheses should imply P. A proof is then a tree of sequents. The main goal is placed at the root (bottom) of the tree, and the proof tree is constructed upwards from the root by applying *inference rules* of the form:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge_i \qquad \frac{\Gamma \vdash B \quad \Gamma \vdash B \Rightarrow A}{\Gamma \vdash A}\ \Rightarrow_e$$

$$\frac{\Gamma, B \vdash A \quad \Gamma, C \vdash A \quad \Gamma \vdash B \vee C}{\Gamma \vdash A}\ \vee_e$$

Rule $\Rightarrow_e$ is used to introduce new lemmas, i.e. the proof of A is split into two branches, the proof of B and of $B \Rightarrow A$. Rule $\vee_e$ allows a proof by cases: A is proved assuming B, and assuming C; of course, the completeness of the decomposition into cases has also to be proved

($\Gamma \vdash B \vee C$). The latter sequent will sometimes be noted *completeness?* in the FT-RMS proof tree, for space constraints.

The branches of a complete proof tree should end with axioms. One axiom, which will be noted HYP, states that P is provable under Γ if P is one of the hypotheses of Γ. But, as the FT-RMS proof tree is not complete, we use some convenient notations to denote our subjective assessment of unproved branches: $trivial$ will be used for goals which could easily be proved (e.g., trivial statements of completeness); $assumed$ denotes a part of the proof which was forgotten or considered true by the authors, but which we consider is missing. Additional labels will be used to refer to unclear proof fragments that could not be interpreted in terms of sequent derivation.

The proposed restructuration in natural deduction is much lighter than complete formal reworking. Obviously, it would not be sufficient to establish the correctness of the proof (see e.g. [21]). However, it helps us to get a clear view of the proof structure by explicitly expressing the inference steps, hence guiding the analysis.

5.2 FT-RMS proof in natural deduction

We will now present the detailed analysis of the informal proof of FT-RMS published in [11]. It is based on three intermediate lemmas:

> $[S_1]$: For every task τ_i a slack of at least C_i should be present between kT_i and $(k+1)T_i$.
>
> $[S_2]$: If there is a fault during the execution of task τ_r then the recovery scheme should enable task τ_r to re-execute for a duration C_r before its deadline.
>
> $[S_3]$: When a task re-executes, it should not cause any other task to miss its deadline.

Figure 5 shows how these lemmas are used in the proof structure. The list of hypotheses Γ integrates several notions: the ordering of tasks according to their priorities ($T_1 \leq T_2 \leq \ldots \leq T_n$), the IBRMS feasibility condition, the occurrence of a single fault, and the algorithm itself (slack swapping and recovery scheme). Under these hypotheses, no task instance should miss its deadline, that is, the schedule is feasible. In the proof, the $\Rightarrow_e$ rule is used to replace the main goal by the conjunction of two subgoals $[S_2]$ and $[S_3]$. The subgoals are separetely proved by introducing $[S_1]$ in the hypotheses. The validity proof of the first $\Rightarrow_e$ rule, which should state that the proposed subgoals together imply the main goal, is assumed by the authors. Note that the implication is not as trivial as might be thought at first glance. $[S_2]$ expresses a property to be fulfilled by the faulty task instance, but nothing is said about the fact that any other instance of the same task τ_r should also end before its deadline. Since the re-execution may shift higher priority tasks later in time, and since these higher priority tasks may pre-empt the next instance of τ_r, this would deserve a piece of reasoning. Generally speaking, the fact that $[S_2]$ and $[S_3]$ are formulated in terms of *tasks* – rather than than *task instances* –, may be considered as a lack of precision. The proofs of $[S_1]$, $[S_2]$ and $[S_3]$ are shown in Figures 6 to 8. $[S_2]$ and $[S_3]$ involve a proof by cases, the cases description being given in Table 1.

The proof of $[S_1]$ (Fig. 6) is based on the IBRMS feasibility assumption. Since in IBRMS there is an $U_B\ T_i$ amount of slack inserted within any task period T_i, and since $U_B \geq C_i\ /\ T_i$, then the amount of slack is at least C_i. This proof is obvious if $[S_1]$ does state a property of an IBRMS schedule. However, the comments accompanying the proof suggest us that the authors have a larger interpretation: $[S_1]$ is intended to characterize the amount of time redundancy in an FT-RMS schedule. Indeed, the slack is considered as being swapped with the execution of tasks, and shifted later in time. But the relation between the IBRMS backup slots, and the distribution of idle time in an FT-RMS schedule, is never clearly stated. Hence, we consider that the use of lemmas (1) and (2) to prove $[S_1]$ is not sufficient for the larger interpretation. Note that the confusion between IBRMS and FT-RMS slack was also exposed by the study of Sinha and Suri (see previous section).

The proof of S_2 (Fig. 7) is performed by introducing a sufficient condition, noted *slack within* D_r in the figure: it states that the reserved slack may not be swapped forward to a point beyond the deadline D_r of the faulty instance of τ_r. The justification that the condition is sufficient involves the fuzzy relation between IBRMS and FT-RMS already commented on. The proof of the condition itself is decomposed into three cases. Each of them considers one task τ_h having higher priority than τ_r, and trying to pre-empt it (see Table 1). In the first case, τ_h has a deadline D_h shorter than, or equal to D_r, and pre-empts either the execution or the re-execution of τ_r. The two other cases correspond to τ_h having a deadline strictly greater than D_r and arriving respectively after and before E_r. In the proof tree, we did not develop the proofs of the cases. This is so because we reach the limits of an informal formulation. As an example, the proof of Case 1 (τ_r is preempted and $D_h \leq D_r$) is reproduced below:

> "Due to the recovery scheme *RS*, τ_h will preempt τ_r and the reserved slack is swapped with the execution time of τ_h. Thus, the swapped slack lies within D_h. Since $D_h \leq D_r$ we conclude that any slack swapped with τ_h lies within D_r."

A fine analysis of this proof would require formal definitions for notions such as *recovery scheme*, *reserved slack*, *swapped slack* and *lies within*. Furthermore, some inference steps, such as "Thus" in the second sentence, would have to be stated more clearly. Yet some comments can be made

$$\dfrac{\text{cf. Fig 6}\quad \Gamma \vdash [S_1] \qquad \dfrac{\dfrac{\text{cf. Fig 7}}{\Gamma, [S_1] \vdash [S_2]} \quad \dfrac{\text{cf. Fig 8}}{\Gamma, [S_1] \vdash [S_3]}}{\Gamma, [S_1] \vdash [S_2] \wedge [S_3]}}{\Gamma \vdash [S_1] \Rightarrow [S_2] \wedge [S_3]} \qquad \dfrac{\dfrac{}{assumed}}{\Gamma \vdash [S_2] \wedge [S_3] \Rightarrow \text{feasible schedule}}}{\Gamma \vdash [S_2] \wedge [S_3]} \qquad \Gamma \vdash \text{feasible schedule} \Rightarrow_e$$

Figure 5. Informal Proof Tree of the FT-RMS

$$\dfrac{\dfrac{\dfrac{\text{HYP}}{\Gamma \vdash \forall i.\, slack(T_i) = U_B T_i} \quad \dfrac{\text{HYP}}{\Gamma \vdash \forall i.\, U_B \geq C_i \,/\, T_i}}{\Gamma \vdash (1)\, \forall i.\, slack(T_i) = U_B T_i \;\wedge (2)\, \forall i.\, U_B \geq C_i \,/\, T_i} \qquad \dfrac{\dfrac{}{assumed}}{\Gamma \vdash (1) \wedge (2) \Rightarrow [S_1]}}{\Gamma \vdash [S_1]} \Rightarrow_e$$

Figure 6. Informal Proof Tree of Subgoal $[S_1]$

$$\dfrac{Case1 \quad Case2 \quad Case3 \quad \dfrac{\dfrac{}{assumed}}{\text{completeness?}}}{\Gamma, [S_1] \vdash \text{slack within} D_r} \qquad \dfrac{\dfrac{}{assumed}}{\Gamma, [S_1] \vdash \text{slack within} D_r \Rightarrow [S_2]}}{\Gamma, [S_1] \vdash [S_2]} \Rightarrow_e$$

Figure 7. Informal Proof Tree of Subgoal $[S_2]$

$$\dfrac{\dfrac{Case4 \quad Case5 \quad Case6 \quad \dfrac{\dfrac{}{assumed}}{\text{completeness?}}}{\Gamma, [S_1], i > r \vdash \tau_i \text{ ends before deadline}} \qquad \dfrac{\dfrac{Case7}{\tau_i \text{ delayed}} \quad \dfrac{\dfrac{trivial}{\neg \tau_i \text{ delayed}} \quad \dfrac{\dfrac{trivial}{\text{completeness?}}}{}}{}}{\Gamma, [S_1], i < r \vdash \tau_i \text{ ends before deadline}} \qquad \dfrac{\dfrac{}{trivial}}{\text{completeness?}}}{\Gamma, [S_1], i \neq r \vdash \tau_i \text{ ends before deadline}} \vee_e$$

$$\Gamma, [S_1] \vdash [S_3]$$

Figure 8. Informal Proof Tree of Subgoal $[S_3]$

Table 1. Cases extracted from the informal demonstration

τ_r is the re-executing task, τ_h is a *higher priority* task, τ_l is a *lower priority* task
R_i, E_i and D_i are respectively the ready time, end of execution and deadline of task τ_i

Proof Cases	**Proof Subgoal**	**Priority τ_i wrt τ_r**	**Cases Description**	
1	$[S_2]$	higher	τ_h preempts τ_r	$D_h \leq D_r$ & $R_r < E_h <$ end re-execution
2	$[S_2]$	higher	τ_h is pushed forward in time by the re-execution of τ_r	$D_h > D_r$ & $E_r \leq R_h <$ end re-execution
3	$[S_2]$	higher	τ_h preempts the execution of τ_r	$D_h > D_r$ & $R_r < R_h < E_r$
4	$[S_3]$	lower	τ_r is overlaping τ_l and τ_r is ready before τ_l	$R_r < R_l$ & $D_r < D_l$
5	$[S_3]$	lower	τ_r is completely contained within τ_l	$R_r \geq R_l$ & $D_r \leq D_l$
6	$[S_3]$	lower	τ_r is overlaping τ_l and τ_r is ready after τ_l	$R_r > R_l$ & $D_r > D_l$
7	$[S_3]$	higher	the re-execution of τ_r preempts τ_h	$D_h > D_r$ & $E_r \leq R_h <$ end re-execution

on the chosen decomposition. Each case involves only two tasks, τ_r and τ_h. Multiple pre-emptions involving several higher priority tasks, or several instances of one of higher priority task, are not explicitly taken into account.
Lemma $[S_3]$ (Fig. 8) is probably the hardest subgoal to prove, and actually we know that it is wrong, since Sinha and Suri exhibited a counterexample. $[S_3]$ can be written as: $\forall\, i.\, (\, i \neq r \Rightarrow \tau_i$ ends before its deadline $)$. After a step putting $i \neq r$ into the hypotheses, the proof is divided into two subproofs according to the priority of τ_i compared to τ_r. The first subproof focuses on lower priority tasks. It is done by decomposing the state space into three cases (4 to 6), depending on how the task period (from ready time to deadline) overlaps with the period of τ_r. Once again, the analysis of each case involves only two tasks. It is focused on the direct effect the re-execution of τ_r may have on one lower priority task, by reasoning in terms of slack usage and displacement of the task. But, ignoring the presence of other tasks, it fails to consider more indirect chains of effects: for example, the delayed lower priority task may be pre-empted by another task, after the end of τ_r's re-execution, hence being further delayed. This is precisely what happens in the class of scenarios found by Sinha and Suri. The second subproof focuses on higher priority tasks. After a piece of reasoning, the authors conclude that the only case worth considering is the one causing a higher priority task to be delayed by the re-execution of τ_r (case 7). As previously, the proof of case 7 ignores indirect effects.

5.3 Feedback of the analysis

The analysis allows us to conclude that most of the leaves of the proof tree do not provide strong evidence for correctness. This is not due to trivial logical problems related to specific proof steps. Rather, we found some "meta" flaws in the reasoning that affect the *entire* structure of the proof:

- There is a failure to properly acknowledge the difference between IBRMS (static schedule with fixed backup slots) and FT-RMS (dynamic usage of slack).
- The several proofs dealing with deadlines limit their scope to two task instances (the faulty instance of τ_r, and one instance of a different task).

This led the authors to miss some multiple interaction schemes, which precisely make the complexity of schedulability analysis.

The identification of these weaknesses does not provide us with constructive information for the purpose of testing. Still, there is something that can be extracted from the proof: the authors identified a number of cases (Table 1) which they considered as relevant for verifying the behavior of the algorithm. These proof cases may constitute functional cases to be covered during testing. From the previous analysis, we know that this test criterion is likely to be an imperfect one. We will nevertheless have the opportunity to study whether the functional cases can – to some extent – be related to the incorrect behavior of the algorithm, and whether their imperfection can be compensated by requiring that each case is exercised several times with different test data [25].

6 Test experiments

We first use a blind testing approach (Section 6.1) with random task sets, in order to assess the difficulty of revealing the design flaws in FT-RMS. We also study whether

Table 2. Results of the three random test sets

Fault Distribution	Number of Inputs	Execution Time	Failures Version 1	Failures Version 2	Failure Rate Version 2
i0	10^2	1s	46	5	5%
i1	10^3	5s	141	35	3.5%
i2	10^4	1h30	93	4	0.04%

structural analysis of the algorithm may be useful to improve the test effectiveness. Section 6.2 is devoted to testing the various cases extracted from the informal proof. We investigate the connection of these cases with revealing test inputs.

6.1 Blind testing approach

In order to test the FT-RMS algorithm, we implemented a prototype in C language for each of the two versions of FT-RMS [10, 11] described in Section 3. Each test input is a task set description including:

- the number n of tasks,
- the period T_i and execution time C_i of each task,
- the faulty task τ_r which has to be re-executed (task id and instance number).

We use discrete time, i.e. all input parameters are integer values. Time $t = 0$ is assumed a critical instant (all the tasks are ready). In response to an input, the FT-RMS prototypes compute the corresponding schedule and report the deadline problems.

The IBRMS feasibility condition is integrated in the random generation of task sets to reproduce the assumptions made in the informal proof. The construction of one task set proceeds by adding new tasks as long as the set remains IBRMS schedulable. Each individual task τ_i is such that $5 \leq T_i \leq 130$, $1 \leq C_i < T_i$ and $C_i/T_i < 0.4$. There must be at least two tasks in a set. Since the LCM of the $T_i s$ may be very large, we had to consider a bounded time window: the FT-RMS prototypes actually compute a partial schedule, from $t = 0$ to $t = D_r + T_{n-1} + T_n$, where the deadline D_r of the faulty task instance is no later than $t = 2^{20}$.

We experimented with three variants of the random input distribution; the difference being in the choice of the faulty task instance to be re-executed. Distribution `i0` systematically chooses the first instance of the task having the longest C_i. In Distribution `i1`, the faulty instance is still the first one but the task is now chosen at random. Distribution `i2` chooses the faulty task and instance at random. It is worth noting that `i0` and `i1` impose the fault to occur near the critical instant, while `i2` makes it possible to combine the fault with any configuration of task activity. Table 2 shows the results supplied by test sets generated according to the three distributions.

The observed failures of the algorithm do not reveal new flaws, compared to the flaws already discovered by others. As can be seen, the residual flaw in the second version of the algorithm is much harder to reveal than the one affecting the first version only. In particular, Distribution `i2` turns out to be very poor, since the observed failure rate of Version 2 under this profile is 0.04%. Focusing on the critical instant, and forcing the faulty task to be the one with the longest execution time, allow the efficiency to be increased. However, there is no guaranty that `i0` and `i1` distributions are sufficient to reveal every potential design flaw in the algorithm, since they exclude many fault configurations.

It is well known that blind random testing generally fails to properly exercise the functionalities of a program and its structure. We wanted to see whether the poor results of `i2` could be explained in terms of insufficient coverage of the structure of the algorithm. We used the `tcov` Unix facility to study the branch coverage supplied for the C prototype implementing the second version of FT-RMS (path coverage analysis was intractable, due to the huge number of paths). The results are presented in Table 3. The three test sets supply 100% branch coverage. In the case of the `i2` set, the least covered branch is activated 2242 times. Hence, structural coverage is not the explanatory factor for the poor results of this set.

Table 3. Structural coverage of the C prototype, Version 2 of FT-RMS

Test set	Number of activations of the least covered branch
i0	100
i1	347
i2	2242

In conclusion, these first experimental results confirm that the flaw in the second version of FT-RMS is difficult to uncover using a blind testing approach. We also showed that the use of a classical structural testing approach, namely branch testing, is also inefficient, even if every branch is required to be covered a large number of times. Obviously, testing has to incorporate some functional information about FT-RMS. Focusing on the critical instant in `i0` and `i1` was a crude attempt to do so, but there is no formal justification that the algorithm could not fail under other

fault configurations. We will now see whether the functional cases extracted from the informal proof may be useful to improve the efficiency of testing.

6.2 Coverage of the functional cases extracted from the proof

We had to instrument the C prototype of the second version of the algorithm in order to be able to collect measures of coverage of the functional cases. Note that the first version of the algorithm was not further studied, since the informal proof concerns the second version only.

The collected cases are summarized in Table 4. There are small discrepancies compared to the original cases in Table 1 (end of Section 5). Firstly, Case 1 of the proof was split in two, in order to study separately the preemption of the execution (Case 1a) and re-execution (Case 1b) of τ_r by a higher priority task. Secondly, it turns out that some cases extracted from the informal proof are equivalent, hence they are grouped and detected as a single case during test experiments. Equivalence is obvious for Cases 2 and 7. As for Cases 4 and 6, if a task overlaps τ_r and starts before it (Case 6), its next instance will necessarily overlap τ_r and start after it (Case 4). In fact, Cases 4 and 6 were treated separately in the proof of $[S_3]$ because multiple instances of a same task were not taken into account in the analysis. Finally, we have two additional cases: *unknown higher* (resp. *unknown lower*) reports the presence of at least one task having priority higher (resp. lower) than the re-executing τ_r, and not falling into any of the related proof cases.

Table 4. Experimental cases

Experimental Cases	Proof Cases
1a	1
1b	1
2(7)	2 and 7
3	3
unknown higher	neither 1(a or b), nor 2(7), nor 3
4(6)	4 and 6
5	5
unknown lower	neither 4(6), nor 5

Table 5 shows the result of functional coverage analysis for the three previous test sets. The collected measures give the number of inputs activating the various cases. It is worth noting that the `i2` set had the ability to cover two cases that are never activated by `i0` and `i1` sets, namely *unknown higher* and Case 4. Actually, both cases have a zero probability of occurrence under these distributions. *Unknown higher* is covered if there is at least one task such as none of its instances directly interacts with the faulty instance of τ_r. But, at the critical instant, the execution of τ_r is preempted by all higher priority tasks, if any. As regards Case 4, it requires that the time period of the faulty instance of τ_r overlaps the time period of at least one lower priority task instance. It is easy to see that overlapping is not possible after the critical instant, since the first period of τ_r is fully contained within the first period of any lower priority task.

Table 5. Number of test inputs activating the functional cases

Cases	Test Set `i0`	Test Set `i1`	Test Set `i2`
1a	98	762	5284
1b	38	209	2563
2 (7)	79	251	1825
3	5	40	863
4 (6)	0	0	6089
5	38	761	4631
unknown higher	0	0	3540
unknown lower	0	0	0

The poor efficiency of the `i2` set, having only 0.04% revealing inputs, cannot be explained in terms of insufficient coverage of the functional cases. Indeed, this set supplied adequate coverage, the least covered case (Case 3) being activated by 863 inputs. In order to further investigate the possible correlation between the functional cases and the incorrect behavior of FT-RMS, we performed a detailed analysis of the coverage supplied by the revealing inputs. Table 6 shows the result of the analysis for `i0` and `i2` sets (the 35 revealing inputs in `i1` set are not reproduced for space constraints). It is not possible to establish a precise relation between the triggering of an incorrect behavior, and the coverage of specific (combination of) cases. Every case of the informal proof is covered by at least one revealing output. Combination of Cases 1a and 5 is often observed, but it is neither a sufficient condition, nor a necessary one (as exemplified by the 1733th input of the `i2` set).

Table 6. Case coverage supplied by revealing inputs

Test Set `i0`	Covered cases
schedule 20	1a, 1b, 5
schedule 60	1a, 1b, 5
schedule 71	1a, 2(7), 3, 5
schedule 73	1a, 1b, 5
schedule 96	1a, 1b, 2(7), 5

(a)

Test Set `i2`	Covered cases
schedule 1083	1a, 4(6)
schedule 1733	4(6), *unknown higher*
schedule 3762	1a, 5
schedule 9095	1a, 5

(b)

We conclude that the informal proof of FT-RMS does not provide us with useful information for improving the efficiency of testing. The flaw that was not caught by the

informal proof may also be missed by testing even if a large number of activations is required for each functional case extracted from the proof. This is so, because indirect interaction schemes were ignored in the decomposition into cases, and the incorrect behavior turns out to involve very specific cases of indirect interaction. To illustrate the problem of indirect interaction, let us take the example of one revealing input of the `i2` set (Fig. 9). A transient fault affects the 88th instance of τ_2. Two functional cases are covered by this scenario: Case 1a arises from τ_2^{88} being delayed by a higher priority task (τ_1^{92}), while Case 5 is due to τ_2^{88}'s period being fully contained within the period of a lower priority task (τ_3^{88}). The other task instances play no role in the coverage of functional cases. This is so because the decomposition focuses on pairs of task instances: the faulty instance, and one instance of a different task interacting with it. In particular, the 93rd instance of τ_1 is ignored, since it does not interact with the execution or re-execution of τ_2. Still, τ_1^{93} may interfere with the execution of tasks delayed by τ_2 (as is the case for τ_3), and contribute to the incorrect behavior of the algorithm. In the example, an incorrect behavior is observed because τ_3 was already delayed by τ_1^{92}, and by the execution and re-execution of τ_2, so that it cannot afford an additional delay of C_1. As shown by our test results, there is a low probability of obtaining such a scenario just by chance. Proper worst case analysis would be needed, and the proposed functional decomposition fails to meet the need.

τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	21	107	0.196
τ_2	29	112	0.259
τ_3	29	112	0.259
			$U_B = 0.259$

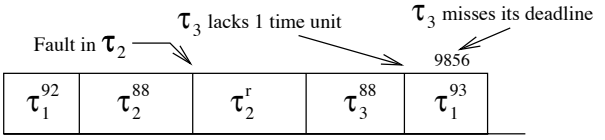

Figure 9. Test Set `i2`, schedule 9095, failure example

7 Conclusions and future work

This paper reports on preliminary work investigating whether proof analysis may enhance the design of testing. In order to concretize this general idea, we took the example of the FT-RMS algorithm and studied whether the structure of its informal proof could be a useful guide for test data selection.

Experimental results have then been obtained by testing C prototypes of the algorithm. These results tend to demonstrate that an informal proof based on intuition is not necessarily helpful for the design of testing, as the test cases extracted from the FT-RMS proof were too loosely connected with the residual fault. But it is not possible to generalize such a statement from a single case study. Structural testing was also ineffective to reveal the fault, and dynamic scheduling problems are known to be hard to test. Furthermore, it might be the case that the proof is simply too weak to carry relevant information for testing: as discussed in Section 5, the FT-RMS proof suffers from major problems affecting its entire structure. In such cases, it seems useless to further consider the proof for the design of testing.

Careful reading is likely to be insufficient to decide whether or not an existing paper proof is worth being considered for testing purposes. We used Gentzen's natural deduction in sequent style to restructure the informal discourse. The resulting proof tree was a convenient compact representation of the proof structure, allowing us to analyze it step by step. We found several flaws in this proof. Some were related to the loose semantics of the notion of slack, used for both IBRMS and FT-RMS schedules. Others were due to the incompleteness of the proof cases, which focused on *direct* interactions with the re-executing task, and failed to consider *multiple* interaction schemes. These flaws are more linked to the proof meta-level than to specific proof steps. Still, we experienced that the tree representation was a useful guide to question the soundness of the proof in a systematic manner. Also, the fact that some of the proof fragments could not be interpreted in terms of sequent derivation (see the undeveloped branches of the proof by cases in Figures 7 and 8) was a good indicator of problems of imprecise definitions, and of doubtful short cuts in the reasoning. Hence, we believe that such a light analysis approach (compared to complete formal reworking) should be sufficient to make a quick assessment of the degree of rigorousness of an informal proof.

Building from this, the viability of informal proof analysis for testing purposes will have to be studied on more convincing proof examples. One case study we are currently looking at is the group membership protocol presented in [15]. Its informal proof is much more crafted than the FT-RMS one, because it was consolidated by using model-checking to analyze instances of the protocol. Still, the protocol was found flawed after publication[1]. It will be interesting to see whether this example better supports the idea of testing being guided by the argument for correctness.

It also remains to investigate the design of testing from formal (but partial) proofs. We will gain a formalization of the different notions used in the proof, which will prevent any shift in the semantics of such notions. But being

[1] See the corrected version of the paper downloadable at http://www.csl.sri.com/papers/wdag97

formal does not ensure the usefulness of a partial proof for the purpose of testing. The proof decomposition into subgoals could be irrelevant as well, and the subgoals could prove very hard to link with test cases. This last problem did not appear in the informal proof analysis because the cases, used as lemmas, could be interpreted in terms of program state configurations. But such a problem is very likely to appear with a formal proof. This requires further investigation, which we will conduct by formalizing the FT- RMS problem, based on the work of B. Dutertre on the PCP algorithm [5].

References

[1] P. Ammann, P. Black, and W. Majurski, "Using model checking to generate tests from specifications", in *2nd IEEE Int. Conf. on Formal Engineering Methods (ICFEM'98)*, Brisbane, Australia, IEEE Computer Society, pp. 46–54, Dec. 1998.

[2] B. Cukic, "Combining testing and correctness verification in software reliability assessment", in *2nd IEEE High-Assurance Systems Engineering Workshop (HASE'97)*, Washington, DC, IEEE Computer Society, pp. 182–187, Aug. 1997.

[3] L. Dong, R. Melhem, D. Mossé, S. Ghosh, W. Heimerdinger, and A. Larson, "Implementation of a transient-fault-tolerance scheme on DEOS", in *5th Real-Time Technology and Application Symposium (RTAS'99)*, Vancouver, Canada, IEEE Computer Society, pp. 56–66, June 1999.

[4] B. Dutertre, "Formal analysis of the priority ceiling protocol", in *IEEE Real-Time Systems Symposium (RTSS'00)*, Orlando, FL, pp. 151–160, Nov. 2000.

[5] B. Dutertre and V. Stavridou, "Formal analysis for realtime scheduling", in *19th AIAA/IEEE Digital Avionics Systems Conf. (DASC'00)*, Philadelphia, PA, Oct. 2000.

[6] E. Egan, D. Kutz, D. Mikulin, R. Melhem, and D. Mossé, "Fault-tolerant RT-Mach (FTRT-Mach) and an application to real-time train control", *Software: Practice and Experience*, vol. 29, no. 3, pp. 1–17, 1999.

[7] J. C. Fernandez, C. Jard, T. Jéron, L. Nedelka, and C. Viho, "Using on-the-fly verification techniques for the generation of test suites", in *8th Int. Conf. on Computer-Aided Verification (CAV'96)*, LNCS 1102, Springer-Verlag, pp. 348–359, Aug. 1996.

[8] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications", in *Joint 7th Eur. Software Engineering Conf. and 7th ACM SIGSOFT Int. Symp. on Foundations of Software Eng. (ESEC/FSE'99)*, LNCS 1687, Springer-Verlag, pp. 146–162, Sept. 1998.

[9] M. Geller, "Test data as an aid in proving program correctness", *Communications of the ACM*, vol. 21, no. 5, pp. 368–375, 1978.

[10] S. Ghosh, R. Melhem, and D. Mossé, "Fault-tolerant rate monotonic scheduling", in *6th IFIP Conf. on Dependable Computing for Critical Applications (DCCA'97)*, Garmish-Partenkirchen, Germany, IEEE Computer Society, pp. 121–145, Mar. 1997.

[11] S. Ghosh, R. Melhem, D. Mossé, and J. S. Sarma, "Fault-tolerant rate monotonic scheduling", *Real-Time Systems*, vol. 15, no. 2, pp. 149–181, 1998.

[12] J. B. Goodenough and S. L. Gerhart, "Towards a theory of test data selection", *IEEE Trans. on Software Engineering*, vol. SE-1, no. 2, pp. 156–173, 1975.

[13] S. Hayashi, R. Sumitomo, and K. Shii, "Towards the animation of proofs – testing proofs by examples", *Theoretical Computer Science*, vol. 272, no. 1–2, pp. 177–195, Feb. 2002.

[14] M. Joseph, *Real-Time Systems: Specification, Verification and Analysis*, Prentice Hall, 1996, London.

[15] S. Katz, P. Lincoln, and J. Rushby, "Low-overhead time-triggered group membership", in *11th Int. Workshop on Distributed Algorithms (WDAG'97)*, M. Mavronicolas and P. Tsigas, Eds., LNCS 1320, Springer-Verlag, pp. 155–169, Sept. 1997.

[16] G. Lima and A. Burns, "An effective schedulability analysis for fault-tolerant hard real-time systems", in *13th Euromicro Conf. on Real-Time Systems (ECRTS'01)*, Delft, The Netherlands, pp. 126–135, June 2001.

[17] C. L. Liu and J. Layland, "Scheduling algorithm for multiprogramming in a hard real-time environment", *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[18] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS", *IEEE Trans. on Software Engineering*, vol. 21, no. 2, pp. 107–125, Feb. 1995.

[19] D. J. Richardson and L. A. Clarke, "Partition analysis: a method combining testing and verification", *IEEE Trans. on Software Engineering*, vol. 11, no. 12, pp. 1477–1490, 1985.

[20] J. Rushby, "Integrated formal verification: Using model checking with automated abstraction, invariant generation and theorem proving", in *Theoretical and Practical Aspects of SPIN Model Checking : 5th and 6th Int. SPIN Workshops*, LNCS 1680, Springer-Verlag, pp. 1–11, 1999.

[21] J. Rushby and F. von Henke, "Formal verification of algorithms for critical systems", *IEEE Trans. on Software Engineering*, vol. 19, no. 1, pp. 13–23, Jan. 1993.

[22] V. Rusu and E. Singerman, "On proving safety properties by integrating static analysis, theorem proving and abstraction", in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, LNCS 1579, Springer-Verlag, pp. 178–192, 1999.

[23] P. Sinha and N. Suri, "Identification of test cases using a formal approach", in *29th Symposium on Fault-Tolerant Computing (FTCS'99)*, Madison, WI, IEEE Computer Society, pp. 314–321, 1999.

[24] P. Sinha and N. Suri, "On the use of formal techniques for analysing dependable real-time protocols", in *21st IEEE Real-Time Systems Symposium (RTSS'00)*, Phoenix, AZ, IEEE Computer Society, pp. 126–135, 1999.

[25] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet, "Software statistical testing", in *Predictably Dependable Computing Systems*, H. B. Randell, J-C. Laprie and B. Littlewood, Eds., Springer Verlag, pp. 253–272, 1995.