On the Use of Formal Techniques for Analyzing Dependable Real-Time Protocols*

Purnendu Sinha
ECE Dept.
Boston University
Boston, MA 02215
sinha@bu.edu

Neeraj Suri
Dept. of Computer Engineering
Chalmers University
Göteborg, Sweden
suri@ce.chalmers.se

Abstract

The effective design of composite dependable and real-time protocols entails demonstrating their proof of correctness and, in practice, the efficient delivery of services. We focus on these aspects of correctness and efficiency, specifically considering the real-time aspects where the need is to ensure satisfaction of stringent timing and operational constraints. In this paper we establish the use of mathematically rigorous techniques such as formal methods (FM's) in not only providing for their traditional usage in establishing correctness checks, but also for their capability of assessing and analyzing timing requirements in dependable real-time protocols. We present our perspectives in utilizing FM's in developing exact case analyses of fault-tolerant and real-time protocols. We discuss the insights obtained and flaws identified in the hand analysis over the process of formally analyzing and verifying the correctness of an existing fault-tolerant real-time scheduling protocol.

1 Introduction

Computers used for critical applications utilize dependable and real-time protocols to deliver reliable and timely services. Additionally, apart from the stringent operational and timing considerations there is also a driver to deliver high performance and maximal utilization of system resources. In such systems, the delivery of time-critical services relies on mechanisms, primarily scheduling disciplines, to satisfy operational and timing constraints. However, the design and analysis of optimal scheduling algorithms for real-time systems is known to be computationally intractable. Exacerbating the situation is the fact that correctness proofs for composite dependable and real-time algorithms and their exact-characterization tend to be quite complex and, potentially, error prone. The demonstration of the correctness of composite dependable real-time protocols is usually performed through simulation studies though at a cost of lack of rigorousness and not being comprehensive to cover the entire state-space of the protocol operation. From the viewpoint of establishing correctness and efficient delivery of services, we focus on time-critical aspects of dependable real-time protocols in this paper.

The effective use of formal techniques for analysing *"discrete"* dependable or real-time services has been demonstrated in [4, 5, 11, 12, 14]. In this paper, we investigate formal methods capabilities for rigorous analysis of *"composite"* dependable real-time protocols when the environment is not fully predictable (e.g., systems operating in non-deterministic environments, i.e., in presence of faults). Specifically, our primary contributions in the formal analysis and verification of dependable real-time protocols are:

- We demonstrate the capability of formal methods to be used as an analysis tool by developing the exact case analysis of a fault-tolerant (FT) version of rate monotonic algorithm (RMA), i.e., FT-RMA. We discuss insights obtained over the process of formally specifying and verifying the correctness of RMA and FT-RMA.
- We highlight the capability of formal techniques to identify flaws in the existing (and published) hand analysis of a FT-RMA algorithm, which we encountered while formally specifying and proving its correctness.

The organization of the paper is as follows: Section 2 describes the system model chosen for our proposed formalism and briefly discusses formal methods. Section 3 overviews the rate monotonic algorithm and

*Supported in part by DARPA Grant DABT63-96-C-0044 and NSF CAREER CCR 9896321

its fault-tolerant extension (FT-RMA). Section 4 develops our perspectives on deriving exact-case analysis of FT-RMA. We present the formal specification and verification of scheduling algorithms in Section 5. We conclude with a brief discussion in Section 6 on the capabilities and caveats in the use of formal techniques in the real-time arena.

2 System Models and Formal Methods

For our study, we consider conventionally used systems [8, 2, 3, 10] where the model involves tasks which have specific correlation (e.g., periodicity, utilization, etc.), follow a deterministic order of execution, and with provision for incorporating non-deterministic aspects such as interrupts and faults.

Consider a set of n independent, periodic and preemptible tasks $\tau_1, \tau_2, \cdots, \tau_n$, with periods $T_1 \leq T_2 \leq \cdots \leq T_n$, and execution times $C_1, C_2, \cdots, C_n$, respectively, being executed on a uni-processor system where each task must be completed before the next request for it occurs. All tasks are preemptible, and preemption overhead is assumed to be negligible.

We assume transient, intermittent and permanent faults (fail-stop, fail-silent). Furthermore, we consider [2, 3] that: (a) fault recovery can be carried out by re-execution of the faulty task, (b) two successive faults are separated by an interval of at least $T_n + T_{n-1}$, and (c) errors can be detected and the cost of detection is negligible or can be incorporated in a task's computation time.

Formal methods [4, 11], through their capabilities of deductive reasoning and mathematical induction, provide extensive support for (a) precise and unambiguous specifications and identification of specification level inconsistencies, verifiable design assumptions, and (b) automated and exhaustive state exploration over the verification process. Because of the state-space explosion problem associated with dependable and real-time protocols, we have chosen proof-theoretic[1] formal approaches which utilize logical reasoning and derivations, as well as rules of induction, to obtain a formal proof basis for the desired system operations. We point the reader for details on formal methods to the excellent study in [11]. For our research, we utilize SRI's Prototype Verification System (PVS)[2] tool [9]; although any formal environment based on higher-order logic can be utilized.

[1] Formal methods include model-theoretic and proof-theoretic approaches for verification. We refer the reader to [11, Section 2.2] for a more detailed discussion on these approaches.

[2] PVS is being used both for its public domain availability and for its comprehensive theorem proving environment.

With this background, we now present the classical RMA approach and subsequently its fault tolerant extension, namely, FT-RMA which is subjected to formal specification and verification. We begin with our motivation for the selection of FT real-time scheduling for demonstrating the capabilities of formal techniques for analyzing dependable real-time protocols.

3 FT-RT Scheduling : An Overview

In dependable time-critical systems, tasks must be executed by their respective deadlines in order to deliver desired service despite the occurrence of faults. Although our proposed formal approach is developed as a generalized analysis tool; in order to demonstrate a tangible analysis capability, we have selected the FT rate monotonic algorithms (FT-RMA) as they are representative of a large class of composite dependable, real-time protocols. FT-RMA was developed in DCCA [2] and a modified journal version in [3]. Over the process of using these protocols [2, 3] to show viability of our formal V&V approach [14], we have also been able to identify test cases [13] that actually make the published FT-RMA protocols of [2, 3] fail. We first introduce the RMA [8], which is the basis for FT-RMA [2, 3].

3.1 RMA : A Review

The *Rate Monotonic Algorithm* (RMA) [8] is an optimal static priority algorithm for the task model of Section 2, in which a task with shorter period is given a higher priority than a task with longer period. A schedule is called *feasible* if each task starts after its release time and completes before its deadline. A given set of tasks is said to be *RM-Schedulable* if RMA produces a feasible schedule. The processor utilization of n tasks is given by $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$.

A set of tasks is said to *fully utilize* the processor if (a) the RM-schedule meets all deadlines, and (b) if the execution time of any task is increased, the task set is no longer RM-schedulable. Given n tasks in the task set with execution times C_i for task τ_i; if $C_i = T_{i+1} - T_i \quad \forall i \in \{1, n-1\}$, and $C_n = 2T_1 - T_n$, then under the RMA, the task set fully utilizes the processor.

The following theorem provides a *sufficient* condition to check for RM-schedulability.

Theorem 1 (L&L Bound [8]) *Any set of n periodic tasks is RM-schedulable if the processor utilization is no greater than $n(2^{\frac{1}{n}} - 1)$.* □

This schedulability bound of [8] being rather pessimistic (for large values of n, it converges to 69%) necessitates an exact case analysis of the RMA policy for

its practical applications. The necessary and sufficient conditions for RM-schedulability based on processor utilization appear in [7]. Another approach to provide the necessary and sufficient condition for schedulability is based on worst-case response time calculation [6]. We describe this later approach to compute response times based on the number of invocations in an interval and the computation time needed for each invocation. For simplicity, we assumes a single task of priority j, and all tasks phasings being zero.

The number of events at priority level j arriving in the interval between two time instants x and y is computed as $number(x, y, j) = \lceil y/T_j \rceil - \lceil x/T_j \rceil$. The computation time needed for these invocations is $compute\_time(x, y, j) = number(x, y, j) \times C_j$.

So, at level i the computation time needed for all invocations at levels higher than i is computed as:

$$total\_compute\_time(x, y, i) = \sum_{j=1}^{i-1} compute\_time(x, y, j)$$

Let $R(x, y, i)$ be the response time at level i in the interval $[x, y)$. If $total\_compute\_time(x, y, i)$ is zero, the processor will not be preempted in the interval and the whole of the time will be available for the use at level i. If $total\_compute\_time(x, y, i) > 0$, then the response time at level i cannot be less than $y +$ $total\_compute\_time(x, y, i)$. The above argument can be defined recursively:

```
R(x,y,i) =
if(total_compute_time(x,y,i) = 0) then y
  else R(y, y+total_compute_time(x,y,i), i)
endif
```

The worst-case response time at level i can then be defined as $R_i = R(0, C_i, i)$. If no computation is needed at levels $0, \cdots, i-1$, then the response time at level i is the computation time C_i itself; if not, then the amount of time needed at the higher levels is added to the computation time.

Thus, a necessary and sufficient condition is given as follows:

Theorem 2 (Exact-case Analysis — 2 [6]) *A given task set is schedulable iff* $\forall i\ 1 \leq i \leq n,\ R_i \leq T_i$, *where deadlines* d_i *are bounded by* $C_i \leq d_i \leq T_i$. □

The classical RMA does not have any provision for re-execution of tasks to handle transient fault instances, nor does it provide any guarantees for their backup task schedulability. In the next section, we outline how time-redundancy can be incorporated into RMA to provide guarantees for tasks to meet their deadlines even in the presence of faults [2, 3].

3.2 FT-RMA : A Review

Currently, there are ongoing efforts in extending RMA to provide for fault-tolerance by incorporating temporal redundancy [2, 3, 10]. As mentioned in the beginning of Section 3, we have selected the approach proposed in [2, 3] for our study. This approach describes a recovery scheme for the re-execution of faulty tasks, including a scheme to add slack (i.e., idle time) to the schedule, and further derives schedulability bounds for a set of tasks considering fault-tolerance through re-execution of task. Furthermore, cases with a single or with multiple faults within an interval of length $T_n + T_{n-1}$ are considered. Faults are assumed to be transient such that a single identified faulty task can be re-executed.

Along with guaranteeing the schedulability of a task set in the fault-free case (RMA), we need to ensure the schedulability of a task set even when the backups are used for re-execution of some task. A recovery scheme that ensures re-execution of any task after a fault has been detected must satisfy the following conditions:

S1: There should be sufficient slack for any one instance of any given task to re-execute.

S2: When any instance of τ_i finishes executing, all slack distributed within its period should be available for the re-execution of τ_i in case a fault is detected.

S3: When a task re-executes, it should not cause any other task to miss its deadline.

The original recovery scheme proposed in [2] is: *"The faulty task should re-execute at its own priority."*

The following lemmas show the published [2] proof of correctness of this approach.

Lemma 1 ([2]) *If* $U_B \geq C_i/T_i, \quad i = 1, \ldots, n$, *then* **S1** *is satisfied.* □

Lemma 2 ([2]) *If* **S1** *is satisfied, and swapping*[3] *takes place, then* **S2** *is satisfied.* □

Lemma 3 ([2]) *If* **S1** *and* **S2** *are satisfied, and the faulty task is re-executed at its own priority, then* **S3** *is satisfied.* □

A FT-RMA utilization bound was computed to guarantee schedulability in the presence of a single fault. This schedulability bound was derived as: $U_{FT-RMA} = U_{LL}(1 - U_B)$, where U_B is equal to the maximum of all tasks utilizations ($U_B = \texttt{max}\ U_i$).

[3] The slack is shifted in time by being swapped with the task's execution time if no fault occurs.

However, this recovery scheme of [2] may fail in meeting a task's deadline, even though a given task set satisfies U_{FT-RMA} bound. A modified recovery scheme is presented in [3] as:

> "In the recovery mode, τ_r will re-execute at its own priority, except for the following case: *During recovery mode, any instance of task that has a priority higher than that of τ_r and a deadline greater than that of τ_r will be delayed until recovery is complete.*"

The approaches presented in [2, 3] attempted to provide FT support to a task set satisfying U_{FT-RMA} bound. As mentioned, in [3] the authors have modified the recovery scheme of [2], our initial interest was to explore the capabilities of the formal process to identify the cause due to which a recovery task fails to meet its deadline, and to highlight the shortcomings in arguments presented in the hand analysis of the original version of FT-RMA [2]. Concurrently, this also resulted in developing the exact case analysis of FT-RMA, which we describe in the next section.

4 Our Perspectives on Exact-Case Analysis of FT-RMA

The results presented in Section 3.2 provide sufficient conditions to check for schedulability under fault assumptions. From a practical standpoint, the utilization bound U_{FT-RMA} is quite pessimistic arguing for exact-case analysis of FT-RMA. In this section, we present such an exact-case analysis, with the recovery scheme in which the faulty task re-executes at its own priority. We extend the procedure outlined in Section 3.1 (Theorem 2) to incorporate the re-execution time for the faulty task.

When a fault is detected during execution of a task, the task needs to be re-executed. It is assumed that the task re-executes at the end of its original scheduled time-slot; in order to ensure that the recovering task will meet its deadline, we need to compute the worst case response time of this re-executing task and compare this value with its deadline. As depicted in Fig. 1, the release time of this recovering task is considered to be the task's worst case response time in the original schedule. Let WC_m^i denote the worst case response time of the m^{th} instance of task i, i.e., $WC_m^i = R(m \times T_i, m \times T_i + C_i, i)$ Then, the condition to guarantee schedulability of recovery task can be expressed as:

$$R(WC_m^i, WC_m^i + C_i, i) \leq (m+1) \times T_i \qquad (1)$$

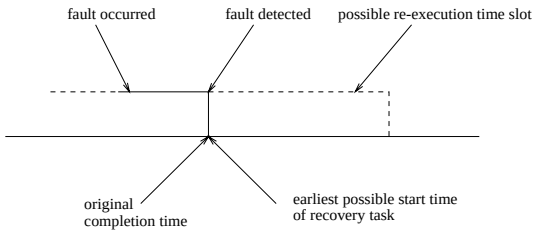

Figure 1: Re-execution of a Faulty Task

Since the faulty task re-executes at its own priority, it will not affect higher priority tasks. To confirm that lower priority tasks are not affected by a faulty task, τ_i, being re-executed, we need to compute the worst case response times of these lower priority tasks (w.r.t. τ_i) incorporating re-execution time of τ_i, and compare them with their respective deadlines.

Let $EST(j)$ be the earliest start time of a lower priority task τ_j relative to recovery task τ_r, where $r < j \leq n$. The execution time of τ_j will be at least delayed by C_r. Incorporating this factor into the response time calculation, the condition to guarantee schedulability of lower priority tasks (w.r.t. τ_r) is given as:

$$R(EST(j) + C_r, EST(j) + C_r + C_j, j) \leq T_j \qquad (2)$$

Theorem 3 (Exact-case Analysis — 3) *A task set is schedulable under single fault assumption iff Equations 1 and 2 are satisfied.* □

The above statement readily follows from our earlier discussions.

We have provided an exact characterization for a set of periodic tasks scheduled by the rate monotonic algorithm for a single fault case. This exact characterization can be used to determine whether a recovery task would meet its deadline or not, thereby ensuring schedulability of a given task set. We present the formalization of the exact characterization in Section 5.2.

5 Formalization of FT-RT Protocols : (RMA → FT-RMA)

An ability to formally specify and verify a given fault-tolerant real-time protocol is an essential element in our approach. Along with the formal aspect, we will also discuss what interpretation/information we were able to deduce from the formal approach. The formal specification and verification of algorithms are performed using PVS[4]. The main effort in formal specification was devoted in formalizing various assumptions

[4]For comprehensive PVS references, the reader is referred to http://pvs.csl.sri.com.

on task and system models, system requirements, the scheduling policies, fault assumptions, and recovery schemes and associated conditions they satisfy. The key idea in our formal verification is to demonstrate the consistency of proof of correctness of any algorithm by transforming it into a mathematical (or calculational) activity that can be easily checked by a mechanical theorem prover. We present formal specifications in the following sections, 5.1 and 5.2. Although the PVS syntax for RMA and FT-RMA is used in the subsequent sections, we will explain the nuances as necessary for the discussion. In the formal representation below, we will identify the associated definitions, axioms and theorems of RMA and FT-RMA which were earlier presented in Sections 3.1 and 3.2. These are marked as % in the PVS specifications.

5.1 Formal Specification of RMA

The formal specification[5] of RMA is embodied in the PVS theory called *rma*. `Task_ID` is defined as a predicate subtype of *posnat*. The types `posreal` and `Real_Time` are defined to be subtypes of *real*. The types `period_range` and `execution_range` are defined as subtypes of `posreal`. `Task_State` is an enumerated type. The task characteristics are defined as `Property` which is a record type with fields `Period, Execution, Phasing` and `State` which are of types `period_range, execution_range, Real_Time` and `Task_State`, respectively. Our aim here is to enumerate over the exhaustive task sets satisfying various timing constraints and a given utilization bound.

```
Property : TYPE = [# Period : period_range,
  Execution : execution_range,
  Phasing: Real_Time, State : Task_State #]
```

`Task_Vector` is defined as an array with index type `Task_ID` and element type `Property`. The constraints on task's execution time, periodicity, deadline and priority are declared as `AXIOMS`. All other assumptions for RMA are declared as dependent types and definitions in the specification.

```
Constraint_AX : AXIOM FORALL (i:Task_ID):
       Execution(T(i)) <= Period(T(i))

Instance : TYPE = nat
Occurrence:[Task_ID, Instance -> Real_Time]=
   LAMBDA (i:Task_ID, j:Instance):
           Phasing(T(i)) + j * Period(T(i))

Release:[Task_ID, Instance -> Real_Time]=
   LAMBDA (i:Task_ID, j:Instance):
           Occurrence(i,j)

Periodic_LM : LEMMA
(FORALL (i:Task_ID, j:Instance) :
   Occurrence(i, j+1) - Occurrence(i, j) =
   Period(T(i)))

%Deadline being the next occurrence of task
Deadline : [Task_ID, Instance -> Real_Time]
Deadline_AX : AXIOM
    (FORALL (i:Task_ID, j:Instance):
        Deadline(i, j) = Occurrence(i,j+1))

% Distinct priorities for tasks
Priority : [Task_ID -> posreal]
Priority_AX:AXIOM (FORALL i : NOT EXISTS j:
               Priority(i) = Priority(j))

% Priority assignment by RMA
 K : posnat
 RMA_Priority : [Task_ID -> posreal] =
 LAMBDA (i : Task_ID) : K / Period(T(i))

% Definition of Rate Monotonic priority
RMA_AX : AXIOM (FORALL i : FORALL j :
    Period(T(i)) > Period(T(j)) IMPLIES
    RMA_Priority(i) < RMA_Priority(j))
```

The processor utilization and a schedulability condition (Theorem 1) are formalized below:

```
% Definiton for Processor Utilization
Utilization : real =
sigma(1, N, LAMBDA i :
      Execution(T(i)) / Period(T(i)))

% Sufficient Condition for schedulability
% (Theorem 1) U <= N * (root(2, N)-1).
RMA : Conjecture
 2 >= exponent((1 + Utilization/ N), N)
```

Following these definitions, we now discuss formal representation of response time calculation described in Section 3.1. For simplicity, we assume a single task of priority j, and all tasks phasings being zero. The response time calculation is specified below:

```
% Number of events at priority level "j"
% in [x, y) time-interval
number(x, y : Real_Time, j : posnat):int =
 ceil(y/Period(T(j))) - ceil(x/Period(T(j)))
```

[5] The complete specifications for RMA and FT-RMA are at http://eng.bu.edu/~suri/specs/specs.html.

```
% Computation time for events at level "j"
compute_time(x, y : Real_Time, j : posnat):
 Real_Time = number(x, y, j)*Execution(T(j))

% Computation time for all invocations at
% levels higher than "i"
total_compute_time(x,y:Real_Time, i:posnat):
 Real_Time = sigma(1, i-1,
   (LAMBDA (i : int) : compute_time(x,y,i)))

% Response time at level i in [x, y)
x, y : var Real_Time
pr : var posnat
 Response(x, y, pr) : recursive Real_Time =
 (IF total_compute_time(x, y, pr) = 0 THEN y
     ELSE Response(y, y +
          total_compute_time(x, y, pr), pr)
  ENDIF) Measure (LAMBDA x, y, pr : x)
```

For example, to check whether a given task set is RM-Schedulable or not for the case where all tasks phasings are zero, we can apply the response time calculation (Theorem 2) as below:

```
RM_Schedulable?(T: Task_Vector): bool =
(FORALL (i:rng):Period(T(i))<=Period(T(i+1)))
 AND (FORALL (i: Task_ID):
 Response(0,Execution(T(i)),i)<=Period(T(i)))
```

The predicate `RM_Schedulable?` captures the notion that if tasks are ordered as per their period and if task response times are less than the respective periods then the task set is RM-schedulable.

A set of tasks is said to *fully utilize* the processor if (a) the RM-schedule meets all deadlines, and (b) if the execution time of any task is increased, the task set is no longer RM-Schedulable. We specify this as:

```
Incr : real = 1
Fully_Utilize?(T: Task_Vector): bool =
RM_Schedulable?(T) AND (EXISTS (i: Task_ID):
 EXISTS (j : Task_ID) : NOT
 Response(0,Execution(T WITH [(i):=(T(i) WITH
 [Execution := Execution(T(i))+Incr])](j)),j)
      < Period(T(j)))
```

This formalization tests if a given task set fully utilizes the processor. Besides validating the task set to be RM-schedulable, it checks for the existence of a task whose response time exceeds its period if the execution time of any other task in the set is increased.

At this stage we have formalized the basic features of the RMA on which the FT-RMA is developed. In the next section, we formalize the assertions provided in the hand-analysis of the FT-RMA [2].

5.2 Formalization of FT-RMA and Exact-Case Analysis

The formal specification of FT-RMA is embodied in a PVS theory called *ftrma*. The pvs theory *rma* is explicitly imported, as the theory *ftrma* builds/extends upon the rate monotonic theory. We begin with formalizing the calculation of the backup utilization U_B which is defined as $\max(U_i)\ \forall i = 1, \ldots, n$. This is computed by using `max(U, n)` function which finds the maximum value of task utilization from an array U of size n. The backup time or slack within a period L is calculated as:

```
Backup(L: posreal) :  posreal = UB * L.
```

Next, we attempt to formalize the computation of backup slot length between the m^{th} period of τ_i and n^{th} period of τ_j. We need to check that beginning period of any other task does not exist between times mT_i and nT_j. This can be specified as follows:

```
boundary?(i,j:Task_ID, m,n:Instance): bool=
 NOT EXISTS(k : Task_ID, l : Instance) :
   m * Period(T(i)) < l * Period(T(k)) AND
    l * Period(T(k)) < n * Period(T(j))
```

The predicate `boundary?` is true if there does not exist any time lT_k such that $mT_i \leq lT_k \leq nT_j$ hold, i.e., there is no beginning period of l^{th} instance of task τ_k within times mT_i and nT_j. Now, the backup slot calculation between mT_i and nT_j is specified as:

```
BS(i,j: Task_ID, m,n: Instance): real =
 (IF boundary?(i,j,m,n) THEN
  UB * (n * Period(T(j)) - m * Period(T(i)))
  ELSE 0 ENDIF)
```

As discussed in Section 3.1, the worst case execution time of any instance of a task can be computed as follows:

```
WCET (i: Task_ID, m: Instance): Real_Time =
 Response(m*Period(T(i)),
          m*Period(T(i))+ Execution(T(i)), i)
```

We attempt to formalize the assumption that two faults are separated by the interval of length $T_n + T_{n-1}$.

```
Fault_AX: AXIOM
(FORALL(i,j:Task_ID, l,m:Instance):
 faulty(i,l) AND faulty(j,m) AND
 NOT EXISTS (k : Task_ID, n : Instance):
 (faulty(k,n) AND Release(k,n)<=Release(j,m)
 AND Release(k,n) >= Release(i,l)) IMPLIES
 WCET(j,m) >=
 WCET(i,l)+(Period(T(N))+Period(T(N-1))))
```

The above axiom states that for any two consecutive faults in a task set, the worst case execution times of the two faulty tasks are separated by at least $T_n + T_{n-1}$.

The sufficient condition, U_{FT-RMA}, for schedulability under a single fault assumption, $U_{LL}(1 - U_B)$, is formally specified as:

```
FT-RMA : Conjecture (2 >=
 exponent((1 + Utilization/(N*(1-UB))), N))
```

This basically checks whether a given task set satisfies the fault-tolerant schedulability bound.

Exact-Case Analysis through the Formal Verification Process

In this section, we demonstrate that the exact-case analysis results as a natural consequence of the computational capabilities of formal verification process.

In order to confirm that a faulty task being re-executed will meet its deadline, we need to check that the task's worst case response time, assumed to be released at time being its worst case response time in the fault-free case and being re-executed at its own priority, is less than its period. We formalize this through the predicate `reexecute?` which checks whether a task being re-executed will meet its deadline. (*Refer to Eq. 1 and the discussion on exact-case analysis in Section 4*)

```
reexecute?(i: Task_ID, m: Instance): bool =
 Response(WCET(i,m), WCET(i,m) +
 Execution(T(i)), i) <= (m+1) * Period(T(i))
```

We now formalize various conditions a recovery scheme needs to satisfy to ensure successful re-execution of faulty task. The condition **S1** that slack available for an instance of recovery task, say τ_i, should be at least C_i is formalized as the predicate `slack?`:

```
slack?(n: Task_ID, m: Instance): bool =
 Backup(Period(T(n))) <= Execution(T(n))
```

The condition **S2** states that if there is a fault during the execution, then the recovery task τ_i must be re-executed for a duration of C_i before its deadline. The predicate `recovery?` validates this condition.

```
recovery?(i: Task_ID, m: Instance): bool =
   slack?(i,m) AND reexecute?(i,m)
```

The condition **S3** ensures that if a task re-executes, then it should not cause any other task to miss its deadline. Since the recovery task re-executes at its own priority and rate monotonic order is followed, it can only affect lower priority tasks. To ensure that lower priority tasks do not miss their deadlines, we need to compute the worst case execution time of these tasks, incorporating the re-execution time of the recovery task. This can be expressed exactly as follows:

Let $EST(j)$ be the earliest start time of a lower priority task τ_j relative to recovery task τ_r, where $r < j \leq n$. The execution time of τ_j's will be delayed by at least C_r. We take this factor into the response time calculation as: $Response(EST(j)+C_r, EST(j)+C_r+C_j, j)$ where $EST(j)$ is the worst case execution time of task $j-1$. The PVS specification of this as follows: (*Refer to Eq. 2 and a discussion on exact-case analysis in Section 4*)

```
EST(j : Task_ID) : Real_Time =
  Response(0, Execution(T(j-1)), j-1)
lower_range(r : int) : TYPE =
      {x : int | x > r AND x <= N}
Recovery_Check : CONJECTURE
 FORALL (j : lower_range(r)) :
  Response(EST(j) + Execution(T(r)), EST(j)+
   Execution(T(r))+Execution(T(j)),j) <=
    Period(T(j))
```

As we will discuss further in Section 5.3, we needed to incorporate the specification for slack length calculation based on number of invocation of different tasks and their execution times. Furthermore, to probe into the highlighted inconsistencies we had to specify the full utilization conditions for a task set into the formal specification. We provide the formal representation of these conditions below:

```
% Slack length calculation:
  Slack(t: Real_Time) : Real_Time =
   t - sigma(1, N, LAMBDA i :
    ceil(t/Period(T(i))) * Execution(T(i)))

% Check for full utilization by each task
Full?(T : Task_Vector) : bool =
 (FORALL (j : Task_ID) : IF (j < N) THEN
  Execution(T(j))=Period(T(j+1))-Period(T(j))
   ELSE Execution(T(j)) =
     2*Period(T(1)) - Period(T(N)) ENDIF)
```

So far we have described the formal representation of the assertions and/or arguments presented in the hand-analysis of FT-RMA. In the next section, we outline our steps towards verification and analysis of FT-RMA and the observations deduced from the formal process of specifying the RMA/FT-RMA properties.

5.3 Identification of Flaws in FT-RMA via the Formal Verification Process

As mentioned in Section 3.2, one of our interests is to explore the use of formal methods to identify the cause of failure of recovery task to satisfy its timing constraints. Prior to getting into the details, we briefly describe our developed formal-methods-based approach for V&V. In [14], we had presented the rationale and approaches for the use of formal methods for validation. We have developed graphical representation structures to encapsulate various verification information and schemes exploring deductive capabilities of formal methods to identify test cases for validation. Particularly, we introduced two data structures, *Inference Tree (IT)* and *Dependency Tree (DT)*, with these structures having capabilities for symbolic execution and query processing, respectively. In [13], we explored the deductive capabilities of our formal approach through a case study of FT-RMA and showed how the different queries being posed and their corresponding inferences at various stages of interactive IT/DT-based process facilitated identification of specific test cases.

Our initial efforts were to ensure that conditions **S1, S2** and **S3** given in Section 3.2 are satisfied by attempting to prove putative theorems reflecting expected behaviors of the protocol operations. It is important to mention that *simply by following the assertions given in the hand analysis* of FT-RMA [2], the initial verification process showed *no* flaws in the arguments being presented. As our approach [14] facilitates incorporation of speculative conditions to explore their implications on the overall working of the protocol, we speculatively imposed condition where, under the U_{FT-RMA} utilization bound, a recovery task misses its deadline by being preempted by other higher priority tasks. The subsequent discussions summarizes specific queries being posed in the DT and corresponding inferences generated over the interactive IT/DT process. For detailed discussion on our formal approach for V&V and the IT/DT usages for FT-RMA, the reader is referred to [14, 13].

Based on the formal representation of backup utilization and backup slot distribution over a specified period, verification of recovery conditions indicated that the condition **S1** is satisfied but the condition **S2** is not. It is important to mention that based on the definition of backup utilization and backup slots length calculations, the verification process confirmed that there was enough slack available in the schedule. This flagged discrepancies in Lemma 2 as the condition **S2** should have been satisfied if there was enough slack reserved in the schedule and swapping had taken place. Furthermore, it also revealed that the backup slots reserved for re-execution may not be available for that purpose, thereby contradicting the statement in Lemma 1. These observations led us to incorporate the specification for slack length calculation based on number of invocation of different tasks and their execution times. Following this step, we were able to ascertain that there is not enough slack available for the faulty task to re-execute. At this stage we have been able to flag the inconsistencies in the FT-RMA, though the conditions due to which these inconsistencies are arising is yet to be determined.

Based on the understanding on the rate monotonic theory, we investigated the issues related to the least natural slack length in the schedule. As discussed in Section 3.1, the schedule would have the least idle time when the task set satisfies the full utilization conditions. It is important to mention that in case of the lowest priority task τ_n being faulty, to be able to re-execute successfully under full utilization condition, its execution time should not exceed $(2T_1 - T_n)/2$. The faulty task re-executes at its own priority while recovering. At this stage, various conditions reflecting full utilization of the processor under the imposed speculative conditions were systematically checked. As we have shown in [13], the failure of the *composite* query representing conditions for full utilization of the processor result in the test cases for validating the proposed approaches of FT-RMA. The following *composite* condition (with $\Delta > 0$, considered as small as possible) essentially checks the feasibility of FT-RMA.

$$\begin{aligned} C_i &= T_{i+1} - T_i, \quad \forall\, i \;\; 1 \le i \le n-1, \\ C_n &= (2T_1 - T_n)/2 + \Delta, \\ & \quad \textit{such that } \sum_i U_i \le U_{FT-RMA} \end{aligned} \tag{3}$$

With this condition, a task set fails to be RM-schedulable under the following two fault conditions: (a) the lowest priority task τ_n is faulty, and (b) the second lowest priority task τ_{n-1} is faulty.

We point out that with this set of conditionals and with the second lowest priority task, τ_{n-1}, being faulty, the modified recovery scheme of [3] *fails* to ensure schedulability of the lowest priority task, τ_n, as will be illustrated in Section 5.4.

5.4 Illustrating the Failure of FT-RMA with the Identified Test Cases

As discussed in the previous section, conditions for full utilization of the processor is a guiding factor to validate the proposed schemes of FT-RMA. Let us

τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	0.4	3.6	0.1111
τ_2	0.5	4	0.125
τ_3	0.9	4.5	0.2
τ_4	0.91	5.4	0.1685

Table 1: A set of 4 periodic tasks

consider[6] a set of 4 periodic tasks, $\{\tau_1, \tau_2, \tau_3, \tau_4\}$, with their respective periods being 3.6, 4, 4.5 and 5.4, and the deadline of each task being equal to its period. Utilizing Eq. 3, the execution times are then computed as shown in Table 1. Thus, the values of U_B, U_{LL} and U_{FT-RMA}, as expressed in Sections 3.1 and 3.2, are 0.2, 0.7568 and 0.6054, respectively. Note that the value of C_4 is upper bounded by the execution time such that the corresponding total processor utilization is equal to U_{FT-RMA}. Thus, the execution time of τ_4, C_4, can have any numerical value[7] satisfying $0.9 < C_4 < 0.9144$. As a test case, we choose C_4 as 0.91. Thus, the total processor utilization by the task set is 0.6046. Since the total processor utilization by this task set is less than U_{FT-RMA} (0.6054), with recovery schemes of [2, 3], a single fault should be tolerated by re-execution of the faulty task.

Let us first consider the fault-free case. Since the total utilization of the processor (0.6046) is less than the least upper bound (0.7568), the task set is RM-schedulable. The resulting schedule without considering backup slots is depicted in Fig. 2. In subsequent timing diagrams of the RM-schedule of the task set, τ_i^j denotes the j^{th} instance of task τ_i.

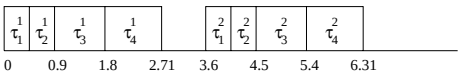

Figure 2: RM-schedule of 4 tasks

We now illustrate the schemes [2, 3] to insert slacks in the schedule by FT-RMA. The backup task can be imagined to be occupying backup slots between every two consecutive period boundaries, where a period boundary is the beginning of any period. Therefore, the length of backup slot between the k^{th} period of τ_i and l^{th} period of τ_j is given by $U_B(lT_j - kT_i)$, where there is no intervening period boundary for any system task. For the given task set with $U_B = 0.2$, the lengths of backup from 0 to T_1 is 0.72, from T_1 to T_2 is 0.08, from T_2 to T_3 is 0.1, from T_3 to T_4 is 0.18, from T_4 to $2T_1$ is 0.36, and so on. The resulting schedule with inserted backup slots is depicted in Fig. 3.

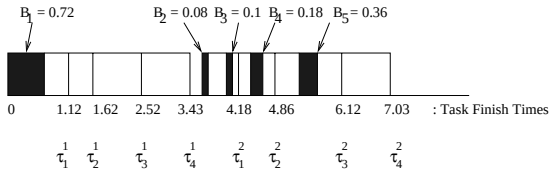

Figure 3: RM-schedule of 4 tasks with backup slots

In the event when no fault has occurred, the backup slots are swapped with the computation time and the resulting schedule would be similar to Fig. 2.

The following two examples [13] highlight the insufficiency of the proposed FT-RMA schedulability bound, U_{FT-RMA}. The first example demonstrates two cases where the original recovery scheme [2] fails to guarantee the schedulability under fault condition, and then the second example highlights a flaw in the modified recovery scheme [3].

Example 1: Two cases where the original recovery scheme [2], the faulty tasks re-executes at its own priority, is found to be flawed.

Case (1) *The lowest priority task, τ_4, misses its deadline if a fault had occurred during its execution and it had re-executed.*

Let τ_4 be a faulty task. τ_1, τ_2, τ_3 and also τ_4 swapped their respective execution time slots with the backup slot B_1. τ_4 finishes at 2.71, and since no other higher priority tasks are ready, it is allowed to re-execute at its own priority. The recovery task τ_4^r only gets to execute for 0.89 time units utilizing backup slot B_1 of length 0.72 time units and a natural slack of length 0.17. During the time interval [3.6, 5.4], the execution of recovery task τ_4^r gets preempted by higher priority tasks and hence, never gets to complete its execution before time 5.4. Fig. 4 illustrates this fact.

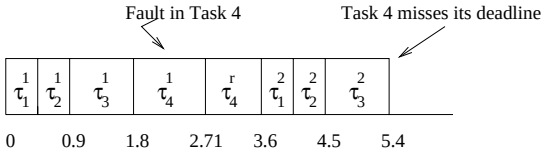

Figure 4: τ_4 misses its deadline

Now referring back to Lemma 1, with backup utilization U_B being 0.2, there exists backup slots of total length 1.08 time units within τ_4's period. As per Lemma 2, with backup slots of length 1.08 time units

[6] It is important to mention that any values for n and periods $T_1, \cdots T_n$ can be considered for illustration purposes, provided the resulting task set satisfies Eq. 3.

[7] The upper bound of C_4 is $\left(U_{FT-RMA} - \sum_{i=1}^{3} C_i/T_i\right) T_4$, which equals 0.9144.

being present and swapping being done, enough slack should have been available for successful re-execution of τ_4, which is not the case here. This is the discrepancy which was highlighted by the verification process.

Case (2) *The lowest priority task misses its deadline due to re-execution of a faulty higher priority task.*

Let τ_3 be a faulty task. As per the recovery scheme, it re-executes at its own priority. The recovery task τ_3^r preempts τ_4, and causes the deadline of τ_4 to be missed. It can be observed from Fig. 5 that τ_4 executes for only 0.9 time units and still would be needing 0.01 time units to complete its execution.

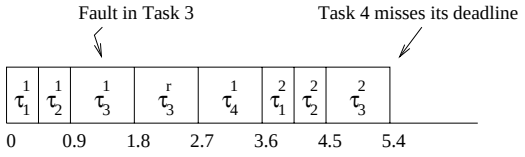

Figure 5: τ_4 misses its deadline

Case (2) highlights the flaw in Lemma 3 where it was proven that a lower priority task would not miss its deadline due to re-execution of a higher priority task. This particular flaw has **not** been discovered by the authors of [2, 3]. However, the case (1) was identified by them, and then based on that observation the recovery scheme was later modified. As we will demonstrate next, the modified recovery scheme is *also* flawed.

Example 2: *A case where the lowest priority task misses its deadline if a fault had occurred in one of higher priority tasks, and the modified recovery scheme [3] has been used for re-execution.*

Consider the same task set as described above. Let τ_3 fail and re-execute at its own priority. This causes τ_4 to miss its deadline. Note that during τ_3's recovery, no other higher priority tasks are ready, therefore, τ_3 would maintain its priority and will complete successfully. As depicted in Fig. 5, τ_4 would utilize backups and execute for 0.9 time units and still would be needing 0.01 time units before time 5.4.

It is important to highlight that we have been able to identify and construct a *specific* task set (see Eq. 3 in Section 5.3) which violates the basis of FT-RMA protocol operations. This case study of FT-RMA has been able to demonstrate that the formal techniques can be used in providing for correctness checks and precise assessment of timing requirements. Furthermore, formal techniques can provide insights into an algorithm/design decisions, and underlying assumptions.

6 Conclusions and Future Work

We have established how formal methods can be used to specify and verify dependable real-time protocols. We have presented a formal analysis of composite fault-tolerant real-time protocols using formal techniques where we have been able to identify flaws in the design analysis. We have discussed several concomitant benefits of the formal approach for analysis of these protocols. We acknowledge that the efficiency and effectiveness of our work will depend on the level of detail used/needed in the abstractions. We are currently investigating issues pertinent to modeling real-time deadlines in the representation techniques. A detailed formal approach to incorporate continuous/dense time model [1] needs to be developed to improve the overall objectives of formal verification and validation process.

References

[1] R. Alur, T.A. Henzinger, "Logics and Models of Real Time : A Survey." *Real Time : Theory in Practice*, (J.W. de Bakker, K. Huizing, W.-P. de Roover, G. Rozenberg, eds.), LNCS 600, Springer-Verlag, pp. 74–106, 1992.

[2] S. Ghosh, R. Melhem, D. Mossé "Fault-Tolerant Rate Monotonic Scheduling." *Proc. of DCCA-6*, 1997.

[3] S. Ghosh, R. Melhem, D. Mossé, J.S. Sarma, "Fault-Tolerant Rate Monotonic Scheduling." *Real-Time Systems*, vol. 15, no. 2, pp. 149–181, Sept. 1998.

[4] C. Heitmeyer, D. Mandrioli, *Formal Methods for Real-Time Computing*. John Wiley, New York, 1996.

[5] F. Jahanian, A.K-L. Mok, "Safety Analysis of Timing Properties in Real-Time Systems." *IEEE Trans. on Software Engineering*, SE 12(9), pp. 890–904, Sept. 1986.

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

[7] J. Lehoczky, L. Sha, Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior." *Proc. of RTSS*, pp. 166–171, 1989.

[8] C.L. Liu, J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." *Journal of the ACM*, 20(1), pp. 46–61, January 1973.

[9] S. Owre, J. Rushby, N. Shankar, F. von Henke, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS." *IEEE Trans. Software Engineering*, SE 21(2), pp. 107–125, Feb. 1995.

[10] M. Pandya, M. Malek,"Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks." *IEEE Trans. on Computers*, 47(10), pp. 1102–1112, Oct. 1998.

[11] J. Rushby, "Formal Methods and the Certification of Critical Systems." *SRI-TR CSL-93-7*, Dec. 1993.

[12] J. Rushby, F. von Henke, "Formal Verification of Algorithms for Critical Systems." *IEEE Trans. on Software Engineering*, SE 19(1), pp. 13–23, Jan. 1993.

[13] P. Sinha, N. Suri, "Identification of Test Cases Using a Formal Approach." *Proc. of FTCS-29*, pp. 314–321, 1999.

[14] N. Suri, P. Sinha, "On the Use of Formal Techniques for Validation." *Proc. of FTCS-28*, pp. 390–399, 1998.