Identification of Test Cases Using a Formal Approach*

Purnendu Sinha and Neeraj Suri
ECE Dept., Boston University, Boston, MA 02215
e-mail: {sinha, suri}@bu.edu

Abstract

A key feature in fault injection (FI) based validation is identifying the relevant test cases to inject. This problem is exacerbated at the protocol level where the lack of detailed fault distributions limits the use of statistical approaches in deriving and estimating the number of test cases to inject. In this paper we develop and demonstrate the capabilities of a formal approach to protocol validation, where the deductive and computational analysis capabilities of formal methods are shown to be able to identify very specific test cases, and analytically identify equivalence classes of test cases.

1 Introduction

Computers that support critical applications utilize composite dependable and real-time protocols to deliver reliable and timely services; the high (and often unacceptable) costs of incurring operational disruptions being a significant design consideration. Due to inherently large state-space covered by these protocols, the conventional verification and validation (V&V) techniques incur prohibitive costs in time needed for their testing. One commonly used validation technique is that of fault injection. Although a wide variety of techniques and tools exist for fault injection [9], the limitations are the actual coverage of the state space to be tested. In this respect, the challenges are to develop a comprehensive and complete suite of test cases over the large operational state space and be able to identify a limited number of specific and realizable tests. Thus, if mechanisms existed that could determine the specific set of conditions (cases) on which the protocol inherently depends, the effectiveness of the overall FI based validation would be significantly enhanced.

Towards these objectives, in [11] we had introduced the use of formal techniques for specification and V&V of dependable protocols, and the process of incorporating implementation information into formal verification. The intent was to utilize formal verification information to aid construct FI experiments for protocol validation. Particularly, we introduced two data structures, *Inference Tree* and *Dependency Tree*, to represent protocol verification information, with these structures having capabilities for symbolic execution and query processing, respectively.

In this paper, we develop our formal approach introduced in [11]. Specifically, we **(a)** explore the deductive and computational analysis capabilities of our formal-method-based query processing mechanisms, **(b)** highlight the capabilities of our approach through a case study of a composite dependable, real-time protocol where we have been able to identify flaws in the analysis, and also ascertain specific test cases, and **(c)** analytically identify equivalence classes of test cases of infinite size.

The organization of the paper is as follows. Section 2 provides a background of our formal approach for pre-injection analysis introduced in [11]. Section 3 overviews the fault-tolerant real-time scheduling protocol that we utilize to demonstrate the effectiveness of our approach. Section 4 outlines our formal approach for identifying specific test cases to validate the protocol under consideration. We conclude with a discussion in Section 5.

2 Formal Pre-Injection Analysis

In [11] we introduced a formal approach for pre-injection analysis to determine fault and activation paths that would guide the FI-based validation of dependable protocols. In this paper, we develop the use of formal techniques identify test cases (pre-injection) to provide a FI toolset for it to construct a FI experiment, i.e., guide the selection and construction of specific FI experiments. We provide a brief review of our basic approach of [11] prior to detailing our test identification process in Sections 3 and 4. We also refer to [11] for a discussion on the impact of refs. [1-6] in the development of our formal approach.

In [11], we developed two novel data structures, *Inference Trees (IT)* and *Dependency Trees (DT)*, to encapsulate protocol attributes generated over the for-

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

mal specification and verification process to identify system states and design/implementation parameters to construct test cases. For both IT and DT, we utilize the fact that fault tolerance protocols are invariably characterized by decision stages leading to branches processing specific fault-handling cases [1, 3, 5, 6, 12]. This is a key concept behind validation, which tries to investigate all the possible combinations of branching over time and with parametric information.

We review the basic features of IT and DT structures prior to discussing their use in identification of test cases in this paper. For a detailed discussion on the IT and DT, we refer the reader to [11].

2.1 Inference Trees: Symbolic Execution

The IT is developed to depict the inference (implication) space involved in a protocol. Each node of the tree represents a primitive *FUNCTION* of the protocol. Associated with each node is a set of *CONDITIONALS* which dictate the flow of operation to the subsequent *ACTION* as defined for the protocol, and the *INFERENCE* space which details the possibility of operations, assertions, and/or usage of event-conditional variables which can be inferred from the operation specification. Fig. 1 depicts an IT for a majority (2/3) voter. We emphasize that the generation of the IT is iterative (see block on top right in Fig. 1).

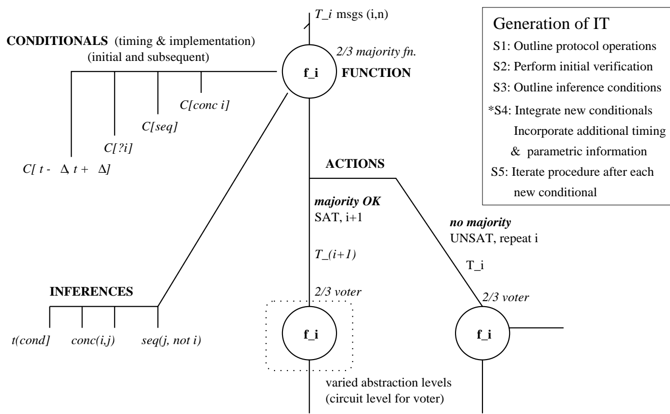

Figure 1: The Inference Tree for a 2/3 Voter Protocol

Although, the IT visually outlines the protocol operations, it does not in itself provide any FI information. The DT structure, described next, utilizes the IT generated inferences to facilitate query mechanisms that get used to identify test cases.

2.2 Dependency Tree: Query Engine

The DT is generated by identifying all functional blocks of a protocol, and ascertaining the set of variables that directly or indirectly influence the protocol operation. Deductive logic used by the verifier is applied to determine the actual dependency (or lack of it) of the function on each individual variable, thus determining the actual subset of variables that influence the protocol operation. Fig. 2 depicts the DT for a multiple round consensus protocol.

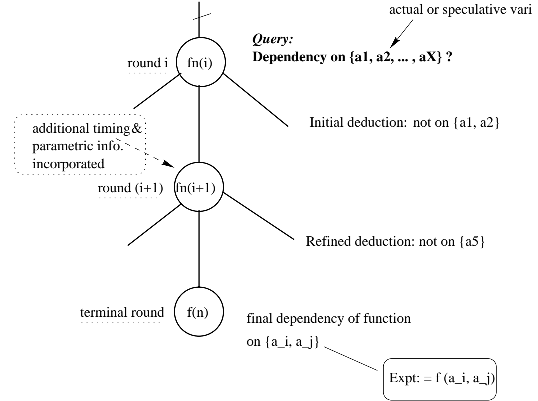

Figure 2: The Dependency Tree : Consensus Example

2.3 Nuances of the IT/DT Approach

The objective of our verification process is to guide the selection of appropriate queries to be posed in the DT. The set of conditionals in the IT are not fixed on *a priori* basis. Each round of iteration can generate constraining conditions which in turn get reflected as new conditionals. This initial set of conditionals serve as an actual (or speculative) list of variables for the DT. At each iteration, the dependency list is pruned as one progresses along a reachability path. In the absence of any new conditionals being added, the dependency list of the DT is monotonically decreasing. In case new conditionals are specified, variables which were pruned earlier from the dependency list may reappear in the next DT iteration.

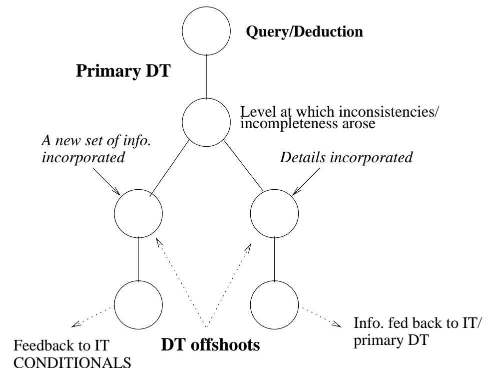

Figure 3: Spawning of the Primary DT

The primary DT represents a given level of specification detail incorporated in the IT. At any stage of query processing if an inconsistency arises, or an incompleteness is found, and accordingly a new set of information is added, the primary DT can have secondary DT offshoots as needed, as illustrated in Fig. 3. The deductions from the spawned DTs are then, as needed, fed back to the parent DT. The overall function dependencies can be used as feedback to specify conditionals in the IT. We emphasize that the DT may not fully represent all possible variable dependencies as it will always be limited to the amount of operational information actually modeled into the formal specification. At any desired level, the elements of the current dependency list provides us with a (possibly) minimal set of parameters which guides formulation of the FI experiments via all permutations and combinations, and *ideally* should generate specific (or a family of) test cases. We repeat that our intent is pre-injection analysis in identifying specific test cases. The actual FI experiments are implemented from these test cases based on the chosen FI toolset(s).

We stress that the IT/DT approach strengthens both verification and validation by making these two processes iterative (over varied implementation detail levels). Fig. 4 represents the general process of FI experimentation using the IT and DT approach.

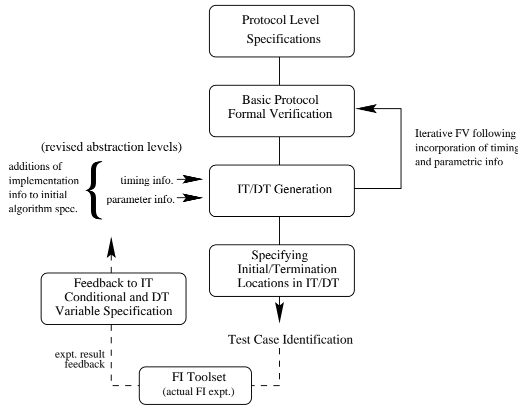

Figure 4: Generating the FI Experiments

The following steps are utilized in our approach to aid the FI process: **(a)** outline protocol operations and establish formal specification of the protocol, **(b)** perform initial verification to demonstrate that the specification conforms to the system requirements, following this, **(c)** generate the IT/DT utilizing the verification information to enumerate the execution paths and establish the dependency of the operations on the design variables, and **(d)** propagate through the DT to identify and select parameters and/or functional blocks to identify test cases for FI.

With this background, we now elaborate our IT/DT based process of ascertaining specific test cases through a case study.

3 A Case Study : FT-RT Scheduling

We have selected the fault-tolerant 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 [7] and a modified journal version in [8]. Over the process of using these protocols [7, 8] to show viability of our formal V&V approach, we have been able to identify test cases that actually make the FT-RMA protocols of [7, 8] fail. We first introduce the RMA [10] protocol on which FT-RMA [7, 8] is based. Given our space constraints, we refer the reader to [10, 7, 8] for details.

3.1 Rate Monotonic Algorithm

The *Rate Monotonic Algorithm* (RMA) [10] is a fundamental scheduling paradigm. 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, i.e., by its specified period. A task's utilization, U_i, is thus C_i/T_i. The processor utilization of n tasks is given by $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$. The RMA is an optimal static priority algorithm for the described task model, in which *a task with shorter period is given 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.

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 RM algorithm, the task set fully utilizes the processor. The following theorem provides a sufficient condition to check for RM-schedulability.

Theorem 1 (L&L Bound [10]) *Any set of n periodic tasks is RM-schedulable if the processor utilization is no greater than $U_{LL} = n(2^{\frac{1}{n}} - 1)$.* □

The classical RMA does not address the issues of fault tolerance. In the next section, we describe an approach proposed in [7] to provide for fault tolerance by incorporating temporal redundancy into RMA.

3.2 FT Rate Monotonic Algorithm

The FT-RMA approach [7] describes a recovery scheme for the re-execution of faulty tasks, including a scheme to distribute slack (i.e., idle time) in the schedule, and derives schedulability bounds for set of tasks considering fault-tolerance through re-execution of tasks. Cases with a single or 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 by a backup task.

A recovery scheme that ensures re-execution of a task must satisfy the following conditions:

> **C[S1]:** There should be sufficient slack for any one instance of any given task to re-execute.
>
> **C[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.
>
> **C[S3]:** When a task re-executes, it should not cause any other task to miss its deadline.

The recovery scheme proposed in [7] being:
The faulty task should re-execute at its own priority.

The following lemmas show the proof of correctness of this approach.

Lemma 1 ([7]) *If backup task utilization (U_B), $U_B \geq C_i/T_i, \quad i = 1, \ldots, n$, then* **[S1]** *is satisfied.* □

Lemma 2 ([7]) *If* **C[S1]** *is satisfied, and swapping*[1] *takes place, then* **C[S2]** *is satisfied.* □

Lemma 3 ([7]) *If* **C[S1]** *and* **C[S2]** *are satisfied, and the faulty task is re-executed at its own priority, then* **C[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 ($U_B = \mathtt{max}\ U_i$).

However, this recovery scheme of [7] 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 [8] 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.*

After this brief introduction to FT-RMA, we now detail our IT/DT based process for identifying test cases for the V&V of FT-RMA.

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

4 FT-RMA: The Formal V&V Process

We initiated the formal verification of FT-RMA to establish the correctness of the proposed solutions based on the assertions provided in the hand analysis of FT-RMA [7, 8]. It is important to note that the verification process only establishes the *correctness* of assertions, and does *not* by itself identify the explicit cause of a verification inconsistency.

4.1 Verification: Identification of Flaws in FT-Rate Monotonic Algorithm

Our initial step was to formally specify[2] and verify the FT-RMA protocols [7, 8]. Since in [8] the authors had modified the recovery scheme of [7] (see end of Section 3.2), our initial interest was to explore the capability of the formal process to identify a cause due to which a recovery task fails to meet its deadline. The main effort in formal specification was devoted in formalizing various assumptions on task and system models, system requirements, the scheduling policies, fault assumptions, and recovery schemes and associated conditions they must satisfy.

We initiated our efforts towards verification of FT-RMA (i.e., to ensure that conditions $C[S1]$, $C[S2]$ and $C[S3]$ in Section 3.2 are satisfied) by attempting to prove putative theorems reflecting expected behaviors of the protocol operations. With the initial verification and subsequent interactive usages of IT/DT (discussed in the next section), we found out that the scheme of [7] fails to ensure schedulability of lower priority tasks and thereby violates the $C[S3]$ stated in Section 3.2. This particular flaw was **not** discovered earlier by the authors of [7, 8]. With the same conditions being imposed on a task set and permitting changes in the priority of the recovery task, we were also able to discover that the modified recovery scheme [8] *also fails*. The process of identifying the causes behind these flaws appear in the subsequent sections, i.e., the test cases.

4.2 Visualization: IT/DT for FT-RMA

The objective of the formal verification and representation of verification information in the IT structure (Fig. 5) is to guide the selection of appropriate queries to be posed in the DT. It is important to note that the selection and formal representation of queries to be posed is still an interactive process. Automating this process is ongoing work.

The various assumptions on task characteristics, utilization bound, task ordering in the schedule, and the feasibility criteria for the task set are reflected

[2] The complete specifications, and issues pertaining to the automation of the formal processes, for RMA and FT-RMA are at http://eng.bu.edu/~suri/specs/specs.html.

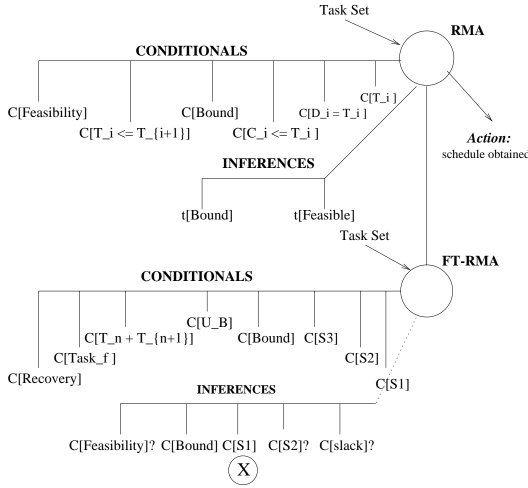

Figure 5: Inference Tree for RMA & FT-RMA

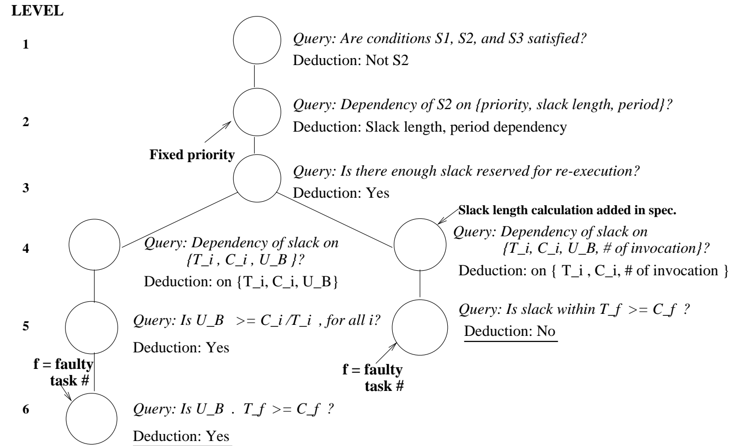

Figure 6: The DT for FT-RMA : Phase I

in the CONDITIONAL[3] space of RMA. Under a no-fault condition (for the given task set) the utilization bound and the feasibility conditions are satisfied, and are indicated in the INFERENCE space. The conditions for successful re-execution of a faulty task, namely, $C[S1], C[S2]$ and $C[S3]$ of Section 3.2, and various conditions on fault-tolerant schedulability bound, backup utilization, time between two faults, faulty task and recovery criteria are specified in the CONDITIONAL space of FT-RMA. The feasibility test under single fault case gets reflected in the INFERENCE space of FT-RMA indicating that the task set meets the U_{FT-RMA} bound but the schedule is not feasible. Based on the formal representation of backup utilization and backup slot distribution over a specified period, verification of recovery conditions also indicated that $C[S1]$ is satisfied but $C[S2]$ is not as indicated by *C[S2]?* in Fig. 5.

The above observation led us to pose queries in our query engine, the DT structure, to identify the exact dependencies of $C[S2]$. During the first phase of query processing in the DT (Fig. 6) at Level 1 we inferred that $C[S2]$ is not satisfied. Further we posed query (at Level 2) to determine the actual dependencies of $C[S2]$ on different parameters. With the priority of recovery task being fixed, the DT deduction declared dependencies on slack length and task's period (as deadline depends on task's period). Next, we posed the query to check whether there is enough slack reserved for the re-execution of the faulty task. Based on the definition of backup utilization and backup slots length calculation, the IT/DT confirmed that there was enough slack available in the schedule. These flagged discrepancies in Lemma 2 as $C[S2]$ should have been satisfied if there was enough slack reserved in the schedule and swapping had taken place. This observation led the primary DT to offshoot two DT's at Level 3 to identify the exact conditions on which satisfaction of $C[S2]$ depends on. The left branch of the DT basically went through the proof of Lemma 1, and as a final deduction indicated that there was enough slack reserved for re-execution of the faulty task. These conflicting observations revealed that the backup slots reserved for re-execution may not be available for that purpose, thereby contradicting the statement in Lemma 1. This information is then reflected in the IT (Fig. 5) as inference $C[S1]$ being marked as X, indicating that as per Lemma 1, $C[S1]$ may not be true. The right branch of the DT incorporated the specification for slack length calculation based on number of invocation of different tasks and their execution times. We next posed the query in the DT to determine whether backup utilization has any effect on the slack length calculation, and it turned out that there is none! We then posed the query, Level 5, to ascertain whether there is slack available in the schedule before the task's deadline. The DT deduced that there is not enough slack available for the faulty task to re-execute. This deduction confirmed that the claim in Lemma 2 is flawed. At this stage the inconsistency in the FT-RMA has been flagged, though the cause behind it is yet to be determined, i.e., the test cases.

4.3 Identification of Specific Test Cases

Observing these discrepancies highlighted by the DT, we started the second phase of the DT — Fig. 7. We incorporated the conditions in the DT to reflect full utilization of the processor by a task set. We

[3] Represented as $C[feasibility], C[Bound]$, etc. in Fig 5

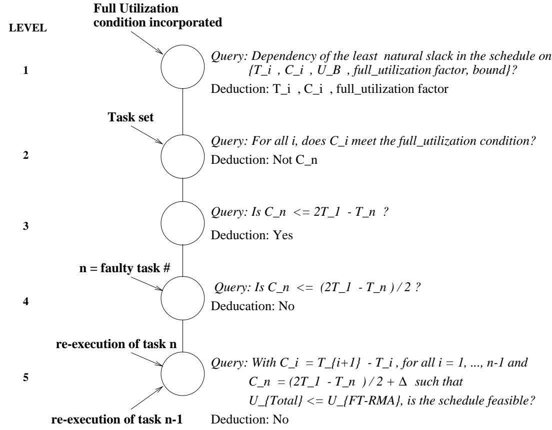

Figure 7: The DT for FT-RMA : Phase II

queried to determine the parameters on which the least natural slack length in the schedule depends on. Next, at Level 2, we posed a query to determine any correlation of the chosen task set to the definition of fully utilized task set. As it turned out that except for the lowest priority task, all other tasks in the set meet the criteria. We then confirmed whether the execution time of the lowest priority task is less than the maximum possible value of C_n. In case of the lowest priority task being faulty, to be able to re-execute successfully under full utilization condition, its execution time should not exceed $(2T_1 - T_n)/2$. At Level 4, the query deduced that this condition is not satisfied for the given task set. With the execution time of the lowest priority task, C_n, being $(2T_1 - T_n)/2 + \Delta$ ($\Delta > 0$ can be considered as small as possible) such that $\sum_i U_i \leq U_{FT-RMA}$, we next posed a query to determine whether such a task set is RM-schedulable under the following two fault conditions: (a) the lowest priority task n is faulty, and (b) the second lowest priority task $n-1$ is faulty. The faulty task is re-executed at its own priority while recovering. We inferred that for the first case the faulty task is not able to re-execute and complete successfully. For the second case, the lowest priority task cannot finish before its deadline due to the re-execution of the second lowest priority task. This led us to conclude that the proofs of Lemma 2 and also Lemma 3 in the hand-analysis failed to consider the case of full utilization of the processor by a task set.

We point out that with these set of conditionals and with the second lowest priority task, τ_{n-1}, being faulty, the modified recovery scheme of [8] fails to ensure schedulability of the lowest priority task, τ_n, as will be illustrated in Section 4.4.

We emphasize that cases to be tested are derived by queries related to discrepancies between the levels. In this case, the discrepancies arose in the first phase of the DT related to the availability of slack for re-execution. Phase II of the DT probed further into this issue. The propagation through the DT (phase II) outlines the set of conditionals those corresponding to full utilization of the processor by a task set which affected the availability of slacks for re-execution of the faulty task. Furthermore, these set of conditionals were enough to pinpoint the insufficiency of the U_{FT-RMA} bound (Level 5). Thus, the failure of query at Level 5 results in this query essentially being the test case, i.e., the test case is:

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

We stress the fact that for validating scheduling protocols, identification of a fault case is similar to identifying a task set which would violate the basis of the protocol operations. We elaborate and illustrate these findings in the following section. Note that this test case will form the basis of constructing a FI experiment using a chosen FI toolset.

4.4 Identified Test Case Effectiveness

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 consider[4] a set of 4 periodic tasks, $\{\tau_1, \tau_2, \tau_3, \tau_4\}$, with their respective periods being 4, 4.5, 5 and 6, and the deadline of each task being equal to its period. Utilizing Eq. 1, the execution times are then computed as shown in Table 4.4. 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[5] satisfying $1 < C_4 < 1.0158$. As a test case, we choose C_4 as 1.01. Thus, the total processor utilization by the task set is 0.6044. Since the total processor utilization by this task set is less than U_{FT-RMA} (0.6054),

[4] 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. 1.

[5] The upper bound of C_4 is $\left(U_{FT-RMA} - \sum_{i=1}^{3} C_i/T_i\right) T_4$, which equals 1.0158.

with recovery schemes of [7, 8], a single fault should be tolerated by re-execution of the faulty task.

τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	0.5	4	0.125
τ_2	0.5	4.5	0.1111
τ_3	1.0	5	0.2
τ_4	1.01	6	0.1683

Let us first consider the fault-free case. The resulting schedule without considering backup slots is depicted in Fig. 8. In subsequent timing diagrams of the RM-schedule of the task set, τ_i^j denotes the jth instance of task τ_i.

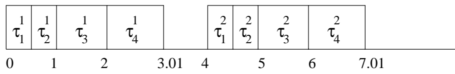

Figure 8: RM-Schedule of 4 tasks

We now illustrate the schemes [7, 8] to distribute slack to the schedule using 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. Thus, the length of backup slot between the kth 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 0.8, from T_1 to T_2 is 0.1, from T_2 to T_3 is 0.1, from T_3 to T_4 is 0.2, from T_4 to $2T_1$ is 0.4, and so on. The resulting schedule with inserted backup slots is depicted in Fig. 9.

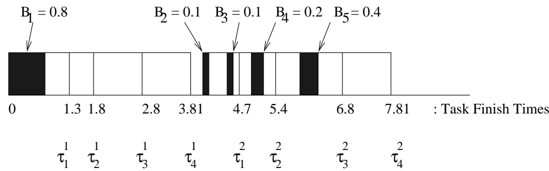

Figure 9: RM-Schedule of 4 tasks with backup slots

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. 8.

Using the identified test case (Eq. 1) derived from the DT, we now illustrate the shortcomings in the recovery schemes of FT-RMA [7, 8]. The first example demonstrates two cases where the original recovery scheme [7] fails to guarantee the schedulability under fault condition, and then the second example highlights a flaw in the modified recovery scheme [8].

Note 1: Two cases where the original recovery scheme [7], the faulty tasks re-executes at its own priority, is found to be flawed.

Case (a) *The lowest priority task 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 3.01, 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 0.99 time units utilizing backup slot B_1 of length 0.8 time units and a natural slack of length 0.19. During the time interval [4, 6], the execution of recovery task τ_4^r gets preempted by higher priority tasks and hence, never gets to complete its execution before time 6. Fig. 10 illustrates this fact.

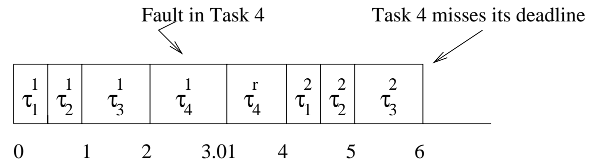

Figure 10: Task 4 misses its deadline

We now relate this to our findings through the IT/DT approach: as per Lemma 1, with backup utilization U_B being 0.2, there exist backup slots of total length 1.2 time units within τ_4's period. Also, per Lemma 2, with backup slots of length 1.2 time units 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 DT queries in Phase I.

Case (b) *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. 11 that τ_4 executes for only 1.0 time units and still would be needing 0.01 time units to complete its execution.

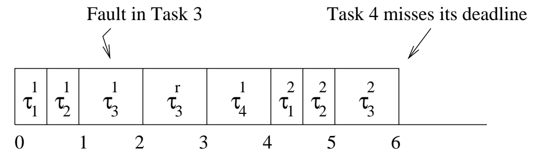

Figure 11: Task 4 misses its deadline

Case (b) 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. Moreover, as we will demonstrate next, the modified recovery scheme is flawed too.

Note 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 [8] 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. 11, τ_4 would utilize backups and execute for 1.0 time units and still would be needing 0.01 time units before time 6.

It is important to mention that the IT/DT based approach enabled us to identify and construct a *specific* case which highlights flaws and inconsistencies in both recovery schemes[7, 8] of FT-RMA.

4.5 Identification of Equivalence Classes

A key idea in FI-based experimental analysis of system dependability is to identify the equivalence class[6] in order to reduce the number of faults to be injected in the system. It was shown in [13] that when the fault population is infinite or extremely large and each fault equivalence class is of finite size, the usefulness of this concept is minimal and may not yield any benefit. In this study, we have identified two equivalence classes: (a) *the lowest priority task in the task set (constructed as per guidelines described in Section 4.3) is the faulty task*, and (b) *the second lowest priority task in the task set is the faulty task*. As shown in Section 4.4, with different values for C_4, we can have an infinite number of task sets generated. Thus, each of our equivalence class has (conceptually) infinite number of fault cases. Moreover, any periodic n-task set satisfying Eq. 1 suffices for any of these equivalence classes.

As a comparative analysis of our technique with conventional approaches, we would like to point out that FT-RMA protocols have been through extensive simulations and random FI, and still these fault cases were not identified. Typically, for simulations, task sets are randomly generated. The execution of all tasks in the set including re-execution of the faulty task is observed for a predetermined length of time (generally, it is taken to be a least common multiple (LCM) of tasks' period). Due to its obvious lacking in considering factors for the full utilization of the processor, a task set thus generated by this method has a low probability that it would belong to one of two equivalence classes. Even if we were able to generate a similar affecting task set, that would belong to one of our equivalence classes.

[6]Ascertaining if specific fault cases are equivalent in their capability of stimulating the system under test.

5 Conclusions and Future Work

We have established how formal techniques can be used to abstract large state space involved in protocols and to guide/supplement the conventional FI approaches. We have demonstrated the effectiveness and efficiency of our IT/DT based approach through an example of FT-RMA where we have been able to identify very specific test cases, and analytically identify equivalence classes of test cases.

A current limitation of our formal approach is the need of an interactive mechanism to effectively pose deductive queries in the DT to obtain a conclusive result. Currently, we are investigating the classes of protocols where the formal approach will be effective in identifying and selecting parameters to construct test cases. We are also automating and interfacing the IT/DT generation and iteration process to other existing FI toolsets [9]. Overall, we believe that we have shown the strength and viability of formal techniques for test case identification.

References

[1] D. Avresky, et al., "Fault Injection for the Formal Testing of Fault Tolerance," *IEEE Trans. on Reliability*, vol. 45, pp. 443–455, 1996.

[2] D.M. Blough, T. Torii, "Fault Injection Based Testing of Fault Tolerant Algorithms in Message Passing Parallel Computers," *Proc. of FTCS-27*, pp. 258–267, 1997.

[3] J. Boué, et al., "MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance," *Proc. of FTCS-28*, pp. 168–173, 1998.

[4] J. Christmansson, P. Santhaman, "Error Injection Aimed at Fault Removal in Fault Tolerance Mechanisms – Criteria for Error Selection Using Field Data on Software Faults," *Proc. of ISSRE*, pp. 175–184, 1996.

[5] K. Echtle, Y. Chen, "Evaluation of Deterministic Fault Injection for Fault-tolerant Protocol Testing," *Proc. of FTCS-21*, pp. 418–425, 1991.

[6] K. Echtle, et al., "Test of Fault Tolerant Systems by Fault Injection," *FTPDS, IEEE Press*, pp. 244–251, 1995.

[7] S. Ghosh, et al., "FT Rate Monotonic Scheduling." *Proc. of DCCA-6*, 1997.

[8] S. Ghosh, et al., "FT Rate Monotonic Scheduling." *Real-Time Systems*, vol. 15, no. 2, pp. 149-181, Sept. 1998.

[9] R. Iyer, and D. Tang, "Experimental Analysis of Computer System Dependability," *Chapter in 'Fault Tolerant Computer System Design'*, Prentice Hall, pp. 282–392, 1996.

[10] 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.

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

[12] T. Tsai, et al., "Path-Based Fault Injection," *Proc. 3rd ISSAT Conf. on R&Q in Design*, pp. 121–125, 1997.

[13] W. Wang, et al., "The Impact of Fault Expansion on the Interval Estimate for Fault Detection Coverage," *Proc. of FTCS-24*, pp. 330–337, 1994.