Formal Analysis of Fault Tolerant Real Time Multiprocessor Allocation and Scheduling Protocols

Nikhil Kumar Varma

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University,
Montreal, Quebec, Canada

July 2004

Your file Votre référence
ISBN: 0-612-94714-9
Our file Notre référence
ISBN: 0-612-94714-9

Abstract

Formal Analysis of Fault Tolerant Real Time Multiprocessor Allocation and Scheduling Protocols

Nikhil Kumar Varma

Dependable real-time distributed systems rely on allocation and scheduling protocols to satisfy stringent resource and timing constraints. As these protocols have both dependability and real-time attributes, verification of such composite services warrants a rigorous and formal levels of assurance for their correctness.

The wide acceptance of formal techniques in the design and development of dependable real-time systems is limited because, most of these formal theories for real-time scheduling have been developed without much regard for their further reuse. This makes the formal specifications and their proof constructs in general difficult to reuse, and to verify or analyze similar or related protocols.

To expand the utility of formal techniques, this thesis explores the possibility of effectively defining and then reusing formal theories in order to simplify verification and analysis for a wide spectrum of dependable real-time protocols.

We present a modular formal analysis of a fault-tolerant version of a real-time task allocation and scheduling policies. The main aim is to develop a library of formal theories for the identified modules for real-time and dependable services which could be systematically, and if required, repeatedly used to develop different and new composite dependable multiprocessor real-time allocation and scheduling protocols.

We demonstrate a rigorous and tool-assisted formal analysis of three multiprocessor real time fault tolerant allocation and scheduling protocols for both periodic and aperiodic task models using the concept of reuasability of previously defined theories. We show the reduced effort in the analysis and verification process by reusing the previously formalized theories. Formal analyses of these protocols have been performed using a mechanized theorem proving environment, called PVS from SRI labs.

Acknowledgements

I would like to thank my advisor Dr. Purnendu Sinha for giving me the oppurtunity to work under his supervision and for every single moment he spent with me.

This thesis takes the present shape because of his guidance at every step. He introduced me to the basics of fault tolerant real time computing and real time scheduling. He has spent a great deal of effort and time to get me introduced to the art of research. I have been supported by him as a friend, philosopher and guide during the course of this research. He has been my idol and I would like to follow his vision and thoughts throughout my life.

I would also like to extend my sincere gratitude to the members from Stanford Research Institute(SRI) labs. They have helped me with technical details in working with the PVS tool.

Nikhil Kumar Varma, July 2004

I dedicate this work to my dear Ma,Papa, my loving Phua, and someone who has recently made inroads to my life and heart.

Contents

List of Figures

Chapter 1

Introduction

In this chapter, we begin with defining real time systems, their characteristics and role in safety/time critical systems. Nowadays, there is a growing trend of use of multiprocessing computing platform for real time applications. In this context, the chapter highlights multiprocessor systems in real time environment. It is evident that real time should adhere to fault tolerant criteria as well. The discussion moves by highlighting dependability in multiprocessor systems from real time perspective. Analyzing these dependable real time protocols is quite challenging, and formal methods in recent years have played a pivotal role in such analysis. We briefly discuss an application of formal techniques for such system analysis and describe related work in that context. The chapter concludes with highlighting observations and contributions made in this thesis.

1.1 An Overview of Real Time Systems

Real time systems are those systems in which correctness of the system depends not only on the logical results of computation but also on the time at which the results are produced [Stan1988].

Real time is a level of computer responsiveness that a user senses as sufficiently immediate or that enables the computer to keep up with some external process. A real-time system consists of a controlling system and a controlled system or plant. The plant is the external environment with which the computer interacts. The controlling system or the computer interacts with the plant using information provided by various sensors. It is important that the information provided by the sensors reflects the actual state of the environment. Hence periodic monitoring of the environment and timely processing of the inputs from the sensors is critical.

A *job* is defined as a unit of work. Examples of job include control-law computation and fast fourier transform. A *task* is defined as a set of related jobs that provide some system function. Examples of a task include data access and data writing in a floppy disk. However, both these definitions are used interchangeably in this thesis.

Every input from the sensors are translated into jobs to be processed by the controlling systems. These jobs or tasks can be *periodic, aperiodic or sporadic* in nature. A periodic task is one that is activated every T time units. An aperiodic task is activated at unpredictable times. A sporadic task is an aperiodic task with an additional constraint that there is a minimum interarrival time between task activations and it typically carries a hard deadline.

Depending on the severity of the consequences that may occur if an imposed timing constraint is not met, real-time applications can be classified as being *hard* or *soft.* In a hard real-time system, a task's failure to meet an imposed timing constraint is considered to be a fatal fault. A hard deadline is imposed on a job because the result produced by the job after the deadline can have disastrous consequences. An example of hard real-time is an air *Supplemental Restraint System* (SRS) for a car. However, late completion of a job in a soft real-time system is not desirable, but a few misses does not cause serious harm. The deadline misses in a soft real-time system would however degrade the performance of the system. An example of soft real-time is ATM machine. Slow processing of a request is annoying, but the result is still correct.

Depending on the consequences of a task failing to meet its timing constraints it can be classified as being either *critical* or *noncritical.* Critical tasks should have timely executions and most of them have hard real-time transactions. Example of critical task can be the interrupt handler, in a real time operating system. Non-critical tasks are usually soft real-time tasks. However to increase system performance these tasks should minimize miss ratio and minimize response time. A task that runs a virus scan in the background can be a noncritical task.

If a work overload occurs such that not all tasks can meet their deadlines, the system scheduler shall shed workload by not executing non-critical tasks so as to minimize the average tardiness of critical tasks. One of the approaches to handle such a situation, where critical tasks can also be guaranteed would be to introduce a multiprocessor computing platform so that such work overload can easily be accomodated.

1.2 Multiprocessor Systems in Real Time Environment

An old saying, "Many hands make the work light ", describes the motivation that leads to the development of multiple-processor systems. At any given time, there is a limitation on the computational capacity with which a single processor can handle a particular task request. That means, a system's workload cannot be handled satisfactorily by a single processor, one solution is to apply multiple processors to the problem.

Multiprocessing Systems is a collection of autonomous computing units interconnected through a communication network or shared memory. The information processing is done through this network or the shared memory.

Today's mission-critical systems make extensive use of multiprocessor computing to share information over large, heterogeneous networks. These mission-critical multiprocessor systems have a real-time infrastructure software systems and high performance networks, which pass information reliably in real-time between many information sources and applications running on different computers.

The present day automobiles use multiprocessor systems to achieve a greater degree of performance and reliability. The latest *PAM-CRASH* distributed memory version is now extensively being used by **BMW** and its suppliers, and has reached production level for crash and safety models. *PAM-CRASH* reduces elapsed time by a factor of 8 for these models, and provides excellent speed-up for baseline models. As a **BMW's** requisite, *PAM-CRASH* distributed memory version offers all *PAM-CRASH* and *PAM-SAFE* solution features with additional flexibility by allocating job in a customized manner in various combinations for 16 processors: for example 1 task runs on 16 processors or 4 tasks run on 4 processors.

1.2.1 Scheduling in Multiprocessor Systems

The standard approach to guarantee a correct runtime behavior for a real-time computing system is to generate a task schedule which maps the application domain to the hardware resource and time domain. The major problem in multiprocessor scheduling is to schedule sufficient amounts of hardware resources, such as processors memory and communication links, to each task in such a manner that the constraints imposed on the system will be fulfilled.

1.2.1.1 Allocation and Scheduling Policies

The process of finding hardware resources for the application tasks is referred to as *task assignment.* Task allocation provides suitable resources for the task using strategies based on requirements on load balancing or utilization. The process of finding time resources for the tasks given a set of hardware resources is called *task scheduling.* The combined process of task assignment and scheduling is referred to as *task allocation.*

Given a scheduling problem, the scheduling algorithm performs a schedulability analysis phase and a runtime configuration phase. For some scheduling techniques, these phases are completely independent while in other cases they are inter-twined and thus function as one single phase. In the schedulability analysis phase, the temporal correctness of the application is verified using abstract models of the architecture and the runtime dispatcher. If the outcome of the analysis shows that the application workload cannot be scheduled under the given conditions, then changes must be made in the chosen application, architecture, runtime dispatcher, and/or real-time scheduler.

In the runtime configuration phase, static information is generated for each task in the system for later use by the runtime dispatcher. The information is derived using abstract models of the architecture and the runtime dispatcher. The amount of information generated in the runtime configuration phase depends on the dispatcher strategy being used. For a time-driven dispatcher, the configuration phase generates a table with explicit start and stop times for a certain number of invocations of each task. For fixed priority dispatching, the configuration phase generates a static priority for each task. In a dynamic priority dispatcher, generation of extra information is not required, since priorities are calculated at runtime.

The present day safety/critical systems are mostly real time systems. The operational behavior of such tasks in the presence of failure are very much warranted. We next discuss fault tolerance issues in real time environment.

1.3 Need for providing Fault Tolerance in Real Time Systems

These days more and more people depend daily on services provided by computer systems. These computers control the commonly used or pervasive systems such as automobiles, elevators, aircraft, banking systems, power plants, and so on. Should these computers fail, the consequences could be disastrous, such as severe economic losses or even the loss of human lives. Since design faults cannot be totally eradicated from such control systems, they will have to be tolerated during operation without a compromise to a great extent on the service.

Fault tolerance is the ability of a system to tolerate faults and continue to execute within the required specifications of times and function. A fault tolerant system is one which continues to perform correctly in the presence of hardware or software faults. Fault tolerance is carried out mainly by error processing and by fault treatment [Ande1982].

Reliability is a measure of the success with which the system conforms to some authoritative specification of its behavior [Aviz2000]. When the behavior of a system deviates from that which is specifed for it, this is called a *failure* [Aviz2000]. In general, less frequent the failures, more reliable is the system. An unexpected internal problem which eventually manifests itself in external behavior as a failure is called an *error* [Aviz2000]. *Fault* is defined as the mechanical or algorithmic cause of an error. A faulty component may result in error under certain circumstances during the lifetime of the system. Thus, in a system a fault causes an error, which in turn may lead to a failure. Since external behavior of a component is the internal behavior of the system, a fault at the system-level is caused by a failure at the component-level.

Error processing aims at removing errors from the state of the system, and fault treatment aims at preventing faults from being activated again. In order to be able to undertake error processing, the system must have detected the error and assessed the damage done by it. Error processing can be done in the following phases [Ande1982, Lee1990, Rand1978]:

1. **Error detection :** In order to tolerate a fault, the primary objective is to detect that an error has occured. In this phase, by means of certain consistency checks it is inferrred whether a fault has occured or not.

2. **Damage assessment :** This phase is cruicial because it would ascertain the extent of the fault that has occured and would also provide pointers to the next phase.

3. **State Restoration :** In this phase the system is recovered to consistent state so that it can continue executing.

4. **Fault treatment (diagnosis and passivation) :** This phase would infer the cause of the fault and isolate the faulty components from future execution.

It follows from the very nature of most real-time applications that there is a stringent requirement for high reliability. The combination of temporal requirements, limited resources, concurrent environmental entities and high reliability requirements, presents the real time system engineer with unique problems. To give high levels of reliability, fault-tolerance is required. Some of the common fault tolerance techniques are:

1. **N-version Software** : In an N-version software system, each module is made with up to N different implementations. Each variant accomplishes the same task, but in a different way. Each version then submits its answer to voter or decider which determines the correct answer, and returns that as the result of the module. This system can overcome the design faults present in most software by relying upon the design diversity concept.

2. **Task Rollback by Checkpointing:** A software bug in one task leading to processor reboot may not be acceptable. A better option in such cases is to isolate the erroneous task and handle the failure at the task level. The task in turn may decide to rollback i.e. start operation from a known or previously saved state.

3. **Slack and Re-execution:** The scheme is based on reserving sufficient *slack* in a schedule such that a task can be *re-executed* before its deadline without compromising guarantees given to other tasks. Only enough slack is reserved in the schedule to guarantee fault tolerance if at most one fault occurs within a time interval.

4. **Primary Backup:** In this model one server acts as the primary with replicas acting as backups. The clients send requests only to the primary server. Whenever the primary server fails, one of the backups becomes primary to provide the fault tolerance.

1.3.1 Dependability in Multiprocessor Systems from Real Time Perspectives

Multiprocessor systems provides an improved *reliability* and *availability.* The replication of data provides a higher availability. If a site or a communication link fails so that one or more sites are inaccessible, it may still possible to access data through replication. Moreover higher reliability is reached because the system is still operable inspite of system crashes or link failures.

In multiprocessor systems, each dependable service which is critical from the real time perspective, is decentralized and performed by different nodes. The results of the different nodes is then directed to a voter which ensures that a minority of faulty nodes is not able to affect the dependability properties of the service. Faulty nodes are expelled from the service and replaced by correctly behaving nodes.

Scheduling comprises of generating a sequence of tasks so as to provide them adequete hardware resource to complete within the allowable time or deadline. We next discuss the scheduling policies in uniprocessor systems.

1.4 An Overview of Uniprocessor Scheduling

In this section, we will discuss the Rate Monotonic Algorithm (RMA), used for scheduling independent periodic tasks on a single processor. A computerized real-time system is required to complete its work on a timely basis. The aim of real-time scheduling is to build-up a sequence of jobs that meets hard timing constraints at run-time. The main characteristic of real-time systems is the behavioral predictability.

1.4.1 Rate Monotonic Scheduling

Rate monotonic analysis was developed for scheduling periodic tasks alone [Liu1973]. A periodic task is defined as a task with a periodic arrival time and a hard relative deadline being immediately preceding the next periodic request of the task.

The fundamental percept of rate monotonic analysis is the assignment of priorities to tasks according to the period with which they occur. The priorities assigned are inversely proportional to the period. Liu and Layland have shown that this scheduling protocol is optimal in the sense that if a task set is schedulable (that is, it is possible to schedule the task set in such a way that all deadlines will be met), then it is also schedulable by the rate monotonic protocol .

Liu and Layland provided the following theorem to test whether or not a task set is schedulable. It should be noted that the bound provided by this theorem is a sufficient condition, hence a task set that fails to meet is bound may still be schedulable.

Theorem 1: [Liu1973] A set of *n* independent, preemptable periodic tasks, with relative deadlines equal to their respective periods, can be feasibly scheduled if

$$\sum_{i=1}^{n} C_i/T_i \le n\left(2^{1/n} - 1\right)$$

where C_i is the execution time required for task *i* and T_i is the period of task *i*.

1.4.1.1 Assumptions of Rate Monotonic Analysis

In order to make the schedulability analysis possible, the following assumptions have been made concerning the task set [Liu1973]:

1. The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.
2. Deadlines consist of run-ability constraints only – i.e., each task must be completed before the next request for it occurs.
3. The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks.
4. Run-time for each task is a constant for that task and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption.

As we mentioned that our interest in this thesis is in multiprocessor environment, we next present the scheduling in multiprocessor environment.

1.5 An Overview of Multiprocessor Scheduling And Allocation

In multiprocessor environment there are several processors available upon which jobs may execute. In such systems, CPU scheduling is part of a broader class of resource allocation problems. The scheduling problem for multiprocessor systems is generally directed to resource allocation for the task to achieve optimality. The most common goal of scheduling is to minimize the expected run time of a task set. Examples of other scheduling criteria include minimizing the cost, minimizing communication delay, and giving priority to certain users processes or needs for specialized hardware devices. The scheduling policy for a multiprocessor system usually involves several of these criteria.

Task assignment is a crucial aspect in multiprocessor scheduling. There are various techniques based on execution time requirements and communication cost requirements. The most common formulations are bin-packing mechanism. The simplest bin packing formulation of task assignment policies assigns tasks to processors according to a uniprocessor scheduling algorithm. There are many other heuristics developed to optimise such allocations.

1.5.1 Extensions to Uniprocessor Scheduling Algorithms for Multiprocessor Environment

The scheduling techniques for uniprocessor systems have to be extended in the multiprocessor environment. This extensions are required because the uniprocessor scheduling techniques do not address issues like task allocations.

An optimal scheduling algorithm on a uniprocessor often is not an optimal algorithm on a multiprocessor. For example, EDF has been shown to be an optimal algorithm, under certain conditions in [Dert1974] for a uniprocessor system. However, in multiprocessor environment, EDF is not the optimal. [Mok1983, Dert1989] have shown that no online scheduling algorithm is optimal on multiprocessor environment, regardless of precedence and mutual exclusion constraints.

The work presented by Funk and Goossens in [Funk2002] discusses the changes that are required in the existing EDF algorithm. They propose a scheduling mechanism based on the notion of processor speed, because in multiprocessing environment the processors may have varying speeds. The notion of processor speed is introducted and new schedules are generated based on that. The work aims search for optimal methods to resolve deadline ties using EDF, i.e., to define EDF tiebreakers. The work concludes with the assertion that no optimal on-line and job-level fixed-priority tiebreaker exists for EDF.

The work presented in [Baru2003] presents a mechanism of using the rate monotonic algorithm for multiprocessing systems. The paper generalizes the model proposed in [Ande2001], in which a *utilization bound* was derived for successfully scheduling tasks using the Rate Monotonic Algorithm on identical multiprocessing platform. The work however does not modify the traditional Rate Monotonic algorithm, but proposes a bound value calculation by which it can be determined whether a task set can be scheduled rate monotonically in the multiprocessor architecture.

These scheduling models involve more complexity, when sporadic tasks are to be scheduled. This is because sporadic tasks are invoked at random order with a minimum inter arrival time making the problem far more complex. The work presented in [Ghos1997b] discusses the scheduling of sporadic tasks by utilizing the concept of overloading.

Analyzing and establishing the correctness of composite dependable and real-time protocols is a problem of growing complexity, and their verification for safety concerns warrants rigorous and strong guarantees of correctness. Formal methods have been used extensively for this purpose. We next present the role of formal approaches in the specification and verfication of scheduling protocols.

1.5.2 Real Time Task Allocation Strategies

In every static multiprocessor system, the tasks are assigned and bound to a single processor. This process is called Task Allocation. Generally the task allocation process is done prior to the task execution. Task allocation process takes into consideration various constraints like processing time requirements of the task and communication costs. In this thesis we take into consideration only the processing time requirements of the task since, we have tasks communicating only via shared memory where the communication overhead is minimal.

The most common task allocation problem is to assign tasks to the available processors such that they are schedulable using a uniprocessor scheduling algorithm. This problem requires simple and good heuristic algorithms to be solved. The most common heuristic algorithm is the First Fit (FF) Algorithm. Considering a task set of $T_1, T_2...T_m$ and processor set as $P_1, P_2...P_n$, the task allocation is done as follows:

- Tasks are assigned on in turn in an arbitrary manner. The first task is assigned to P_1.
- After i-1 tasks have been allocated processors $P_1, P_2...P_{k-1}$, the task T_i is allocated processor P_k only if the task T_i cannot be allocated in processors $P_1, P_2...P_{k-1}$.

The First Fit Decreasing algorithm works on the same principle as FF with the tasks arranged in decreasing order of their respective processor utilizations. All these allocation schemas consider the value $ln2$ as the schedulability criterion or the maximum utilization on each processor. However the scheduling criterion value can be raised by using the rate monotonic utilization ($U_{RM}(n) = n(2^{1/n} - 1)$). This enables more tasks to be allocated on each processor.

The Rate Monotonic First Fit (RMFF) works on this principle where the tasks are arranged rate monotonically and then allocated to processors. The allocation schema works in the similar manner as the FF scheduling algorithm with the only difference being in utilizing the value of U_{RM} to determine the task schedulability on a processor rather than $ln2$.

The Rate Monotonic Next Fit (RMNF) is another heuristic algorithm to allocate tasks in a multiprocessor environment. Considering a task set of $T_1, T_2 ... T_n$ and processor set as $P_1, P_2 ... P_n$, the task allocation is done as follows:

- The tasks are arranged rate monotonically, that is in increasing order of their periods.
- Tasks are assigned in turn the order of their arrangement. The first task is assigned to P_1.
- After i-1 tasks have been allocated processors $P_1, P_2 ... P_{k-1}$, the task T_i is allocated processor P_k only if the task T_i cannot be allocated in processor P_{k-1}.

The RMFF and RMNF allocation algorithms have been used in the protocols discussed as case studies in this thesis.

1.6 Formal Approaches for Specification and Verification of Scheduling Protocols

The correctness of real time scheduling mechanism can be demonstrated using rigorous mathematics. However, the proofs of scheduling algorithms developed are generally more intuitive, than having rigorous arguments resulting in incomplete or flawed proofs for the scheduling mechanisms. Formal methods provide extensive support for establishing the correctness of protocol operation by exploring system states over the formal verification process.

The scheduling model proposed in [Liu1973] are more intuitive than being reasonably proven. The risk of incompleteness and flawed proofs is further aggravated on moving to complex models. Real time systems are generally complex and critical in nature, and not following rigorous methods in verifying them would lead to catastrophic consequences.

A method of demonstrating fault tolerance by a scheduling mechanism is presented in [Ghos1997a]. A mechanism is proposed to tolerate faults by re-execution of the faulty tasks. However not proving the scheduling mechanism rigorously resulted in having a flaw in the scheme, resulting in a new scheduling scheme in [Ghos1998].

The need for formal methods has been advocated in [Sinh1999]. This work highlights the risks invovled in *hand analysis* and demonstrates the use of traceable formulations which help in identifying specification level inconsistencies and design level errors. The design level errors in [Ghos1997a], serves as motivation for the use of such formulations as emphasized in this paper.

There has been a limited use of formal techniques in verification of dependable real-time systems, because of the rigourness involved in using them and there is no regard to reusability of the previously defined theories. We propose to reuse the predefined formal contructs using a modular approach. In this regard, we follow the modular approach.

1.7 Related Work

There has been a considerable effort in the development of modularization in the verification and validation of real time and fault tolerant scheduling protocols. The modularization basically aims at combining components to build up the protocol.

We list the related work in the formal analysis of Real Time and Fault tolerant protocols in the following section.

1.7.1 Formal Techniques in Analysis of Real Time Protocols

1.7.1.1 Formal Analysis of Fixed Priority Real Time Systems

Scheduling theories for fixed priority scheduling have sufficiently matured to enable construction of hard real time systems. Preemptive priority based scheduling prescribes a run-time environment in which tasks are allocated a priority and are dispatched using this attribute. Processes are either in runnable state, in which case they are held in the run queue; delayed, in which case they are placed in the delay queue; or suspended, in which case they are awaiting for an event which may be triggered externally or internally to be executed again.

The work presented in [Kettl1995] develops a formal scheduling model for different PC and workstation architecture using a fixed priority model. The methodology proposes a library of scheduling model templates of unique bus structures. This enables designers to select the model template by identifying the structural properties, whenever a new bus is identified.

In [Yama2003], real-time software is formally specified using a hybrid automata, and then a verification methodology is proposed, for its schedulability using both deductive refinement theory and scheduling theory. Using the proposed methods, real-time software is uniformally and easily specified. The schedulability can be verified at its design stage.

The periodic tasks are modeled by a hybrid automata. A periodic task can be specified if the period, execution time and deadline is determined. The next step is modeling of the preemptive scheduler which allocates resource to the periodic tasks. These specifications of the real time system as a hybrid automaton are transformed into phase transition system, and the refinement is verified over phase transition systems [Yama2002].

The other existing formalizations of fixed priority real time systems are in [Alur1996, Vest2000, Brab1999, Alti2002, Liu1999]. In [Alur1996] the preemptive scheduler is specified as a hybrid automata and the safety and liveness property is verified using model checking. The work presented in [Vest2000] proposes a framework using a restricted hybrid automata. This schedulability verification is demonstrated by verifying the reachability problem of a preemptively scheduled task. In [Brab1999] schedulability of a preemptive scheduler is verified by the means of a timed automata. The time is modeled as the maximum and minimum distance between two events in this model. In [Alti2002] a timed automata with priorities based model is proposed. The scheduling policies are also specified according to the priority policies in the work. The work presented in [Liu1999] proposes to specify and verify real-time software and schedulers by deductive verification based on *Temporal Logic Analyser* (TLA).

In [Dute2000] a complete specification and analysis of the priority ceiling protocol is presented. The formalization demonstrates that a detailed specification enables a thorough verification of all scheduling mechanisms in the priority ceiling protocol. All the specifications and proofs are done using the PVS specification and verification system. This provides high assurance of correctness and ensures a complete and rigorous analysis.

1.7.1.2 Formal Analysis of Dynamic Priority Real Time System

The work in [Yuhu1994] presents an approach for formalizing and proving real time properties of dynamic schedulers. The formal logic used in this work is Duration Calculus. The scheduler works on the Deadline Driven Scheduling Algorithm. The specification and proof of the algorithm are formulated using the Duration Calculus.

The work presented in [Zhan2001] is another formal proof for the Deadline Driven Scheduling Algorithm. The proof follows the same formal analysis method as in [Yuhu1994]. Duration Calculus which provides abstraction for random preemption of the processor is used.

1.8 Observations and Contributions

In our viewpoint, the wide acceptance of formal techniques in the design and development of dependable real-time systems is limited because, most of these formal theories for real-time scheduling have been developed without much regard for their further reuse. This makes the formal specifications and their proof constructs in general difficult to reuse, and to verify or analyze similar or related protocols.

Reusability of previously verified theories to prove the correctness of the new protocols developed has been demonstrated in the works presented in [Dhall1978, Klei1993, Leho1989]. The real time protocols presented in these papers are extensions to the work presented in [Liu1973]. The authors in these extended algorithms extend the previously defined proofs for the scheduling tasks on a hard real time environment to propose feasible scheduling mechanisms on complex scenarios with different task sets.

To expand the utility of formal techniques, this thesis explores the possibility of effectively defining and then reusing formal theories in order to simplify verification and analysis for a wide spectrum of dependable real-time protocols. Specifically, we develop the constructs of Rate Monotonic scheduling technique with other supporting modules involving allocation techniques and fault tolerance techniques to develop an extended algorithm which is fault tolerant in nature and utilizes the rate monotonic technique to schedule the tasks which are allocated by any greedy allocation (first fit or next fit).

Our perspective is to exploit the property of modularity in the fault tolerant protocols and to develop a tool verified library of components which can be reused to develop a new protocol. For example, we identify the components that constitute the Fault Tolerant Rate Monotonic First Fit (FTRMFF) protocol described in [Bert1999]. We then specify and verify these components. We emphasize that this process would then shorten the time to develop a new protocol, say Fault Tolerant Rate Monotonic Next Fit (FTRMNF) by using any allocation technique (such as Next fit in our case).

We also demonstrate the formalization of aperiodic tasks based on a scheduling technique described in [Ghos1997b]. We develop a library of components which build up the scheduling technique for aperiodic tasks.

To support our approach, we use the PVS theorem proving environment [Owre1995]. The PVS tool is used extensively to verify and validate the library components that are used to build up the protocol. As mentioned earlier, the objectives are to develop a library of components that could be combined to form a family of scheduling protocols. The developed PVS library also introduces the basic concepts that are central to real-time scheduling analysis and also increases the confidence level on the correctness of the model by adding a great deal of rigorousness in the verification process.

Our intent is not to demonstrate the working of the protocols described in [Ghos1997b, Bert1999] but to develop a reusable library of verified components which can be utilized to build these and other related protocols. The components are independent in nature and can be used easily in constructing other related protocols. The primary objective hence is to have a extensive suite of formal theories to cover a wider spectrum of scheduling policies in literature.

Our specific contributions in this thesis are:

1. Development of a library of components that can be repeatedly used to build up formal specifications of different type of fault tolerant real time periodic scheduling and allocation protocols for periodic and aperiodic tasks.

2. Demonstrate that the effort involved in analyzing a new protocol belonging to the same class is minimised.

1.9 Outline of the Thesis

The thesis is organized as follows: In Chapter 2 we give a description of the component identification and specification technique. Also, a tutorial on the PVS tool that has been used for the purpose of specification and verification of the components and the protocols is presented. In Chapter 3 we describe the library components for periodic scheduling along with the specification and mechanized verification of the protocol. Chapter 4 discusses and formally describes the building blocks for aperiodic scheduling protocol. Finally we conclude in Chapter 5.

Chapter 2

Proposed Modular Approach for Analyzing Fault Tolerant Real Time Protocols

We have emphasized on the need to develop a library of formally verified components in the previous chapter. In this chapter, we identify components that can be formalized and developed as a library of general theories to support reuse of the specifications and the proof-constructs for fault-tolerant real time resource allocation and scheduling protocols. We then introduce the FTRMFF and an aperiodic fault tolerant scheduling protocol. We also give a brief introduction to the PVS tool which will be used for mechanized verification and validation of the formal constructs and the composed protocol.

2.1 Components of Dependable Real Time Protocols

This thesis emphasizes on building up a reusable library of components for fault tolerant real time scheduling protocols by identifying the primary buiding blocks of such class of protocols. In this section, we introduce and discuss the primary building blocks of dependable real time scheduling protocols.

Protocols which provide distributed and dependable scheduling services can often be formulated using their functional primitives. Any resource allocation and scheduling protocol is developed using a computation model. To provide dependability a fault tolerant model should also exist in the functional primitives. Ideally, a library suite should include theories for the following :

1. System and failure model
2. Scheduling
3. Resource allocation
4. Fault tolerence and recovery

Figure 2.1 describes the general framework for verification and analysis of a fault tolerant real time scheduling protocol.

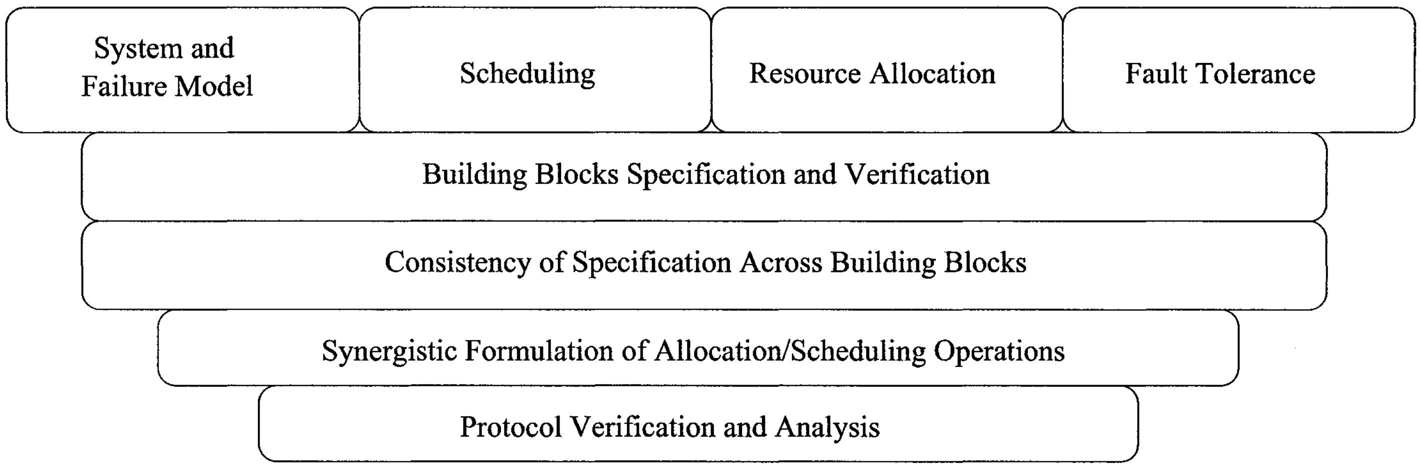
System and Failure Model
Scheduling
Resource Allocation
Fault Tolerance
Building Blocks Specification and Verification
Consistency of Specification Across Building Blocks
Synergistic Formulation of Allocation/Scheduling Operations
Protocol Verification and Analysis

Figure 2.1: General Framework

2.1.1 Basic Building Blocks of Dependable Real Time Scheduling Protocols

In this section we describe the different building blocks for dependable real time scheduling protocols.

1. *a. System model*: We consider a multiprocessor network of processors connected via a shared memory. The system model influences the degree of synchrony to be maintained, i.e synchronous or asynchronous. The degree of synchrony relates to the asumptions made in the time bounds and performance of the system. We illustrate the different synchrony mechanisms by characterizing the functions as modules, [Sinh2001] which is followed throughout this thesis.

   - Synchronous sytems : The attributes for such systems are as follows:

     (a) Every chosen message type is bounded by a delay *d*, which is basically the transmission and processing delay.

(b) Every correct processor have clocks which are monotonically increasing functions of real time.

(c) Every step to be executed by the processor is bounded by an upper time

- Asynchronous systems : The characterization of such systems is done as follows:

  (a) There are no bounds on the message delays of such systems.

In the FTRMFF protocol we model the system as a network of multiprocessor systems which communicate using a shared memory. This model ascertains that there are no communication delays. The task running on each processor are independent from each other and thus there is very minimal communication involved in the system model.

1. *b. Failure model:* A specification of a component or system, baselines the behavior of the component. The specification defines the system responses or transitions on different input sets. The correctness of the system is the consistency of its behavior with the specification.

   A failure model defines the component behavior in the presence of failures. There are various failure models proposed in literature, but we will limit our discussion to fail-stop failure model. In a fail-stop condition it is assumed that the component fails by ceasing execution. We have modeled the fail stop condition in case study of the FTRMFF and the aperiodic scheduling protocols, where the primary backup approach is used to counter such scenarios.

2. *Scheduling* – This component outlines the different scheduling strategies [Liu2000] being employed in the protocol. Scheduling strategy is used to guarantee a cor-

rect runtime behavior for a real-time computing system by generating a task schedule which maps the application domain to the hardware resource and time domain. The strategies can be *static table-driven scheduling*, *static priority-driven preemptive scheduling*, *dynamic planning-based scheduling* and *dynamic best effort scheduling.*

   The static priority driven preemptive scheduling schema is utilized in the implementation done by us in this thesis.

3. *Resource Allocation* – Resource allocation to a task is an important aspect of all multiprocessor scheduling. This component describes the heuristic algorithms [Liu2000, Krish1997] dealing with task assignments to the processors of a multiprocessor system. These include *utilization balancing algorithm*, *first/next-fit algorithm*, *bin-packing algorithm*, *myopic off-line scheduling algorithm*, *focussed addressing and bidding algorithm*, and *buddy strategy.*

   The implementation of the FTRMFF protocol components involves the first fit allocation schema and the FTRMNF involves the next fit allocation.

4. *Fault-tolerance and recovery* – The primary function of all dependable protocols is to provide fault tolerance. This module outlines the approaches employed to provide fault-tolerance. The common approaches fault tolerance and recovery include *N-modular redundancy*, *primary-backup approach*, *(m,k)-firm guarantees*, and *imprecise computation.*

   The fault tolerance in all the scheduling and allocation protocols chosen by us involves the primary backup approach.

After describing the various building blocks for dependable real time scheduling protocols we discuss the various steps to verify and analyze such protocols in the next section.

2.2 Steps for Verification and Analysis of Dependable Real Time Scheduling Protocols

In this section we explain the various steps as shown in fig 2.1 towards verification and analysis of dependable real time scheduling protocols.

2.2.1 Formal Specifi cation of the Building Blocks

So far we have identified the basic building blocks of fault tolerant real time scheduling protocols. The next step is to formally specify each of the components. We use the Prototype Verification System (PVS) tool for this purpose. The component is specified in a higher order logic supported by the PVS tool. Every component has to satisfy a basic property. This property is formulated as a theorem and based on the implementation, a mechanised proof is done by means of theorem proving, to check for the compliance of the component implementation with the specification.

The next step is to compose the components to build up the protocol, and then verify and analyze it, based on the theories which are specified in the components.

2.2.2 Protocol Verifi cation and Analysis

The theories of the basic building blocks can be used to specify the protocol in PVS. The different theories are imported in the final protocol specification and utilized to build up the scheduling protocol. The final protocol has a set of specifications which it has to comply. These specifications are mapped into theorems in PVS and a mechanized proving technique is followed. The PVS theorem prover utilizes the theories and axioms in the various building blocks to prove the theorems of the final protocol. The theorem proving ascertains that the protocol is compliant to the specifications, and would ensure dependable and real time behavior of the implemented protocol.

So far, we have discussed the basic methodology followed by us for protocol verification and analysis. The next section introduces the two protocols which have been used to demonstrate the reuse of component specification in verification and analysis methodology.

2.3 Dependable Real Time Scheduling Protocols

In order to present our methodology of component reuse for verification and analysis of scheduling protocols we have used two different types of scheduling protocols, viz. periodic and aperiodic task scheduling protocols. We present the two protocols in this section.

2.3.1 Dependable Periodic Task Allocation and Scheduling

2.3.1.1 FTRMFF Protocol

The Fault Tolerant Rate Monotonic First Fit (FTRMFF) protocol as described in [Bert1999] has been used to demonstrate the verification and analysis mechanism in this thesis.

The FTRMFF protocol uses a duplication technique in the task schedule. Every task has an active and a passive backup copy assigned to different processors. The active copy of the task is always executed first and whenever a failure occurs the passive copy of the task is executed. This protocol extends the Completion Time Test(CTT) for Rate Monotonic protocol, so as to check the schedulability on a single processor of a task set including backup copies (discussed in chapter 3). It also extends Rate-Monotonic First-Fit assignment algorithm, where all the task copies, including the backup copies, are considered by Rate-Monotonic priority order and assigned to the first processor in which they fit.

2.3.1.2 FTRMNF Protocol

The Fault Tolerant Rate Monotonic Next Fit (FTRMNF) has been conceptualized by us to demonstrate the concept of formalized component reuse, proposed in this thesis. FTRMNF is a dependable scheduling and allocation protocol which uses the rate monotic scheduling princple to schedule tasks based on their periods. This protocol also utilizes the duplication technique as done by the FTRMFF protocols. The primary and backup tasks are scheduled in different processors. The next fit allocation policy is utilized to assign the rate monotonically arranged tasks to available processors. This protocol also utilizes the extended Completion Time Test (CTT) to ascertain the schedulability of a task set on a single processor.

The basic working of the FTRMFF and FTRMNF protocols have been discussed in this section. However the underlying principles and their descriptions will be covered in Chapter 3. We next present a brief introduction to the aperiodic task scheduling protocol.

2.3.2 Dependable Aperiodic Task Scheduling protocol

The scheduling protocol presented in [Ghos1997b] is used to demonstrate the reuse of the building blocks to analyse and verify fault tolerant real time scheduling protocols.

The aperiodic scheduling utilizes a simple slack based dynamic scheduling algorithm as an heuristic to schedule aperiodic tasks. The fault tolerance is provided using the primary backup approach. The primary objective of all aperiodic scheduling protocols is to provide a high aceptance ratio, that is to seek to process most aperiodic tasks. This protocols seeks to achieve it by introducing two techniques, viz. *backup overloading* and *backup deallocation*.

These techniques are introduced so as to reutilize the resources which have been reserved for backup tasks. *Backup overloading* involves scheduling of more than one backup in the same time slot on a processor. *Backup deallocation* on the other hand, reclaims the resource which was reserved by the backup task once the corresponding primary task completes it execution.

It has been demonstrated using various task models that this scheduling schema has a high acceptance ratio of the aperiodic requests and is also dependable in nature.

In the next section, we describe the basic building blocks of the FTRMFF protocol and define their functionality/specification from which their implementation would follow in chapter 3.

2.4 Building Blocks of FTRMFF Protocol

As our focus has been on FTRMFF protocol and Rate Monotonic protocol is the underlying working principle of the protocol, we start by formally characterizing the Rate Monotonic Protocol . We subsequently outline the RMFF and primary-backup routines as components.

2.4.1 Rate Monotonic Component

The underlying principle in the Rate Monotonic scheduling algorithm is defined as follows:

- 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 $C_1, C_2, \cdots, C_n$,repectively, being executed on a uniprocessor system where each task must be completed before the next request for the task occurs.

The scheduling policy is based on assigning a priority to tasks where the priority is inversely proportional to the period of the task. A task with shorter period will have a higher priority as compared to a task with a longer period.

The scheduling is based on the following criterion:

- **Suffi cient Condition** [Liu1973]: Any set of *n* periodic tasks that fully utilizes the processor under RM must have a processor utilization of at least $(n(2^{1/n} - 1))$. The processor utilization of *n* tasks is given by $U = \sum_{i=1}^{n}$.

- **Necessary and Suffi cient Condition** [Leho1989]: Given a set of *n* periodic tasks (with $T_1 \leq T_2 \leq \cdots \leq T_n$). Task τ_i can be feasibly scheduled by RM if and only if

$$\left(\min_{0<t\leq T_i}\left\{\sum_{j=1}^{i} C_j \left\lceil t/T_j \right\rceil /t\right\} \leq 1\right)$$

.

- The entire task set is RM schedulable if and only if

$$\left(\max_{1\leq i\leq n}\ \min_{0<t\leq T_i}\left\{\sum_{j=1}^{i} C_j \left\lceil t/T_j \right\rceil /t\right\} \leq 1\right)$$

The formulation of the RM algorithm is based on the following task model:

1. The request of each task is periodic.

2. All tasks are independent.

3. All tasks are preemptible, and preemption overhead is assumed to be negligible.

4. The deadline of a task is equal to its period.

5. Run-time for the requests of a task is constant for the task.

2.4.2 Rate Monotonic First Fit Component

The Rate Monotonic First-Fit (RMFF) [Dhall1978] algorithm is a partitioning algorithm which allocates tasks to processors following the RM priority order. The tasks assigned to the same processor are then scheduled using RMA.

The scheduling policy in RMFF is based on the RM strategy which can be mathematically described as follows:

- Let $\tau_1, \tau_2, \cdots, \tau_n$ be a set of n tasks with $T_1 \leq T_2 \leq \cdots \leq T_n$. If $C_n/T_n \leq 2\left(1 + u/(n-1)\right)^{-(n-1)} - 1$, where $u = \sum_{i=1}^{n-1} C_i/T_i \leq (n-1)\left(2^{1/(n-1)} - 1\right)$ .then the set can be feasibly scheduled by the RMA.

The allocation policy of this protocol is based on the following formulation:

- With $T_1 \leq T_2 \leq \ldots \leq T_n$, the generic task τ_i is assigned to the first processor P_j such that τ_i and the other tasks already assigned to P_j can be scheduled on P_j according to RM. If no such processor exists, the task is assigned to a new processor.

The RMFF algorithm has the following attributes:

1. The task set has the properties of the task model assumed for the rate monotonic approach (building block 1).
2. The multiprocessor is assumed to consist of identical processors, and tasks are assumed to require no resources other than processor cycles for execution.

2.4.3 Primary Backup Component

Temporal and spatial redundancies are well established approaches for providing dependable services. Conceptually, a chosen task/computation is checked for sanity of results, logical correctness, range checks etc. In case a discrepancy is detected, a backup copy of the task assigned on a different processor gets executed to meet the original task deadline.

The primary-backup component follows the following scheduling policy:

- The primary copy of a task is always executed, while its backup copy β_i is executed according to β_i's status, which can be *active* or *passive*.
- If the status is active, then β_i is always executed, while if it is passive, then β_i is executed only when the primary fails.

The assumption followed in the primary backup component is :

The primary copy τ_i has its request period equal to T_i and its execution time equal to C_i, while the backup copy β_i has the same request period T_i but execution time $C_i' \neq C_i$.

The building blocks which are characterized constructively build up the FTRMFF protocol. The following Figure 2.2 demonstrates the FTRMFF protocol realization from these components.

So far we have characterized the building blocks of the FTRMFF protocol.We next present a discussion on the other two protocols used in this thesis.

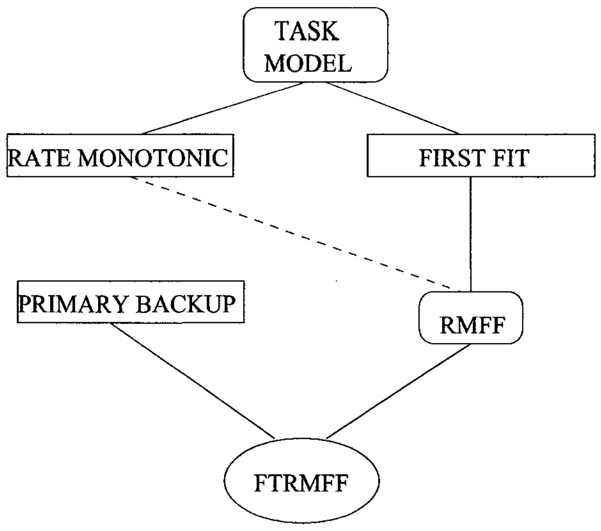

Figure 2.2: Interaction of different modules

2.5 Building Blocks of FTRMNF and Real Time Aperiodic Task Scheduling and Allocation Protocol

The FTRMNF protocol utilizes all the components as the FTRMFF protocol except for the allocation component. The FTRMNF protocol utilizes the Rate Monotonic Next Fit (RMNF) allocation algorithm to allocate the tasks to processors. This component will be discussed in Chapter 3.

The aperiodic scheduling and allocation protocol demonstrated in Chapter 4 also utlizes the components specified for the FTRMFF protocol. We however would like to emphasize that the basic task parameters of periodic and aperiodic tasks are different so a slight modification has been done in the task model to suit the aperiodic task model chosen by us. This will be further discussed in Chapter 4.

We next introduce the PVS tool which has been used for mechanized verification of the dependable real time scheduling protocols.

2.6 Prototype Verification System : An Introduction

PVS is a rigorous formal tool for writing specifications and performing theorem proving activities. There are two classes of automated reasoning systems: one that provides expressive logics, but only limited automation and other with provide extensive automation but a very limited expressive logic. PVS provides automation for a mid-order logic by providing assistance to support clear and abstract specifications sound proofs for difficult theorems.

PVS is a research prototype which captures the state-of-the-art in mechanized formal methods. The PVS system contains:

1. *A parser :* The parser creates the internal abstract representation for the theory described by specification and pops out a message on encountering an error.

2. *A type-checker :* The type-checker checks for semantic errors, such as undeclared names and ambiguous types.

3. *A specification language :* The PVS system follows a high level specification language in which the system is specified.

4. *A theorem prover :* The basic objective of the PVS theorem prover is to generate a proof tree in which all of the leaves are trivially true. The proof tree is generated based on theories that are defined for the system.

The PVS system has many other features. However, a tutorial on PVS is outside the scope of this thesis. We will explain PVS theories in Chapters 3 and 4 with respect to dependable scheduling protocols. We next present a problem specified in PVS.

2.6.1 Specifi cation and Verifi cation in PVS : An Example

This following demonstrates a specification in PVS of the recursive method to find the sum of the squares of the first n natural numbers.
squaresum:

```
squaresum : THEORY
 BEGIN
   number : VAR nat
   squaresump (number) : RECURSIVE nat =

     IF (number = 0) THEN 1
     ELSE number ^ 2 + squaresump(number − 1)
     ENDIF
      MEASURE (λ number : number)
   producttheorem : THEOREM
     squaresump (number) =
      (number × (number + 1) × (2 × number + 1))/6
 END squaresum
```

In this implementation a recursive definition of the squaresum is done. The theorem is specified to verify the correctness of the implementation of the sum of sqares of the first n natural numbers.

So far, we have illustrated the basics of verification and analysis for dependable scheduling protocols. We have introduced the basic building blocks which will be composed to specify these protocols. Our objective is build a library of building blocks for dependable scheduling protocols to reduce the effort in their verification and analysis. We also discussed the building blocks of the FTRMFF protocol in this chapter. In the next chapter we explain the PVS implementation of the building blocks of the FTRMFF protocol and demonstrate that the components can be reused to verify and analyze another periodic scheduling protocols.

Chapter 3

Formal Analysis of Dependable Periodic Task Scheduling and Allocation Protocol

In this chapter, we first present formalization of basic modules identified in Chapter 2. The implemented modules have to comply to certain specification. The specified modules are then utilized to specify the FTRMFF protocol. This protocol is mechanically verified and analyzed by means of proving theorems, which are specified according to the specification requirements of the FTRMFF protocol. Finally, we will illustrate the reuse of the formal specifications to readily facilitate verification and analysis of Fault tolerant Rate Monotonic Next Fit (FTRMNF) protocol.

3.1 Formalization of the Components of FTRMFF Protocol

We formally specify the components of the FTRMFF protocol in PVS. The specification of the components characterizes the working of the components in an abstract manner. We use theorem proving to ascertain the correctness of theories in each of the components.

3.1.1 Formalization of Task Model

We begin with formally specifying the task model that has been used throughout the development of formal specification. At first we define types that are going to be used in formally specifying the attributes of a task set. We define *task_property* as a record type containing task's period, execution time and the phase. This record is assigned to a task ID resulting in a *task_vector* type. We then declare a type to tag a task either as a primary or a backup as well as characterize a backup task as passive or active. Based on these primitives, we define a record type *task_assignment* which for each task ID associates its type (primary or backup) and status as active or passive.

```
task_model: THEORY
  BEGIN
NatNum: posnat
   task_id: TYPE+ = posnat
   TMin, TMax: posreal
   TPeriod: TYPE+ = posreal
   TExecution: TYPE+ = posreal
   task_type: TYPE+ = {primary, backup}
   backup_task_status: TYPE = {active, passive, notbackup}
   TRealTime: TYPE+ = posreal
```

```
PRealTime: TYPE+ = nonneg_real
TaskInstance: TYPE+ = posnat
task_property: TYPE+ =
 [# Period: TPeriod, Execution: TExecution, Phase: PRealTime #]
task_vector: TYPE+ = [task_id → task_property]
task_assignment: TYPE+ =
[# TaskId: task_id,
   TaskType: task_type,
   TaskStatus: backup_task_status #]
```

Next, we declare a *Processor* type. *AllocateProcessor* defines a task assignment on a processor. Another key type definition is that of *Processor_database* which provides a set of tasks assigned on a processor. We also initialize the processor database.

```
Processor: TYPE+ = posnat
AllocateProcessor: TYPE = [task_assignment → Processor]
processor_assignment: TYPE = [set[task_assignment]]
Processor_database: TYPE = [Processor → processor_assignment]
Initialise_database: processor_assignment = emptyset[task_assignment]
```

After having defined these basic types, we next formalize various constraints on task characteristics. As FTRMFF has rate-monotonic underpinnings, we have to impose restrictions on the task to comply with RM task model. These restrictions are imposed by using *axioms.* The implementation imposes the following restrictions:

- The tasks should have a non negative value *(BasicPhase_AX).*
- Period of every task should be positive *(BasicPeriod_AX).*
- The execution time of a task should be less than the period of that task *(ExePeriod_AX).*

- The release time of tasks should be monotonic in nature *(Release_AX)*.
- The deadline of an instance of a task to be equal to the release time of its next occurrence *(Deadline)*.
- The tasks are arranged in a rate monotonic order.

```
AnyTask: task_vector
TaskId1, TaskId2: VAR task_id
Instance1, Instance2: VAR TaskInstance
RTime1, RTime2: VAR TRealTime
BasicPhase_AX: AXIOM ∀ (TaskId1) : Phase(AnyTask(TaskId1)) ≥ 0
BasicPeriod_AX: AXIOM ∀ (TaskId1) : Period(AnyTask(TaskId1)) > 0
Mult_AX: AXIOM ∀ (i: posnat, j, k: PRealTime) : i × j = k
ExePeriod_AX: AXIOM
  ∀ (TaskId1) : Execution(AnyTask(TaskId1)) ≤ Period(AnyTask(TaskId1))
Tphasing: [task_id → PRealTime] = λ (TaskId1) : Phase(AnyTask(TaskId1))
Tperiod: [TaskInstance, task_id → TRealTime] =
    λ (TaskId1, Instance1) : Period(AnyTask(TaskId1))
TOccurence: [task_id, TaskInstance → PRealTime] =
    λ (TaskId1, Instance1) :
      Tphasing(TaskId1) + (Instance1 × Tperiod(TaskId1, Instance1))
Release: [task_id, TaskInstance → PRealTime] =
    λ (TaskId1, Instance1) : TOccurence(TaskId1, Instance1)
Release_AX: AXIOM
  (∀ (TaskId1, Instance1, Instance2, RTime1, RTime2) :
     (Release(TaskId1, Instance1) = RTime1) ∧
      (Release(TaskId1, Instance2) = RTime2) ∧ (RTime1 < RTime2)
      ⊃ Instance1 < Instance2)
Deadline: [task_id, TaskInstance → TRealTime] =
    λ (TaskId1, Instance1) :
      (Release(TaskId1, Instance1) + Period(AnyTask(TaskId1)))
Deadline_AX: AXIOM
```

```
    (∀ (TaskId1, Instance1) :
        Deadline(TaskId1, Instance1) =
          TOccurence(TaskId1, Instance1 + 1))
Period_AX: AXIOM
    ∀ (TaskId1, TaskId2) :
      TaskId1 ≤ TaskId2 ⊃
        (Period(AnyTask(TaskId1)) ≥ Period(AnyTask(TaskId2)))
```

In formally specifying fault-tolerant real-time allocation and scheduling protocols, we utilize some of the basic mathematical functions to carry out the analysis. One such function is the summation of a sequence of natural numbers. These are formalized below:

```
Low, High, Natno: VAR nat
Rno1, Rno2: VAR real
F: VAR [nat → real]
Number: VAR posnat
n: VAR task_id
summation(Low, High, F): RECURSIVE real =
    IF (Low > High)
      THEN 0
    ELSE IF (High = Low)

    THEN F(Low) ELSE (F(High) + summation(Low, High − 1, F)) ENDIF
    ENDIF
     MEASURE (λ Low, High, F: High)
```

3.1.1.1 Completion Time Test

We next formalize the schedulability test called *Completion Time Test (CTT)*. With $T_1 \leq \ldots \leq T_i$ for a set of i tasks ($\tau = \{\tau_1, \ldots \tau_i\}$) which are in phase at time zero, the *cumulative work* on a processor required by tasks in τ during $[0, t]$ is $W(t, \tau) = \Sigma_{\tau_k \in \tau} C_k \lceil t/T_k \rceil$. We create a sequence of time $S_0, S_1, \ldots$ with $S_0 = \Sigma_{\tau_k \in \tau} C_k$, and with $S_{l+1} = W(S_l, \tau)$. If for some l, $S_l = S_{l+1} \leq T_i$, then τ_i is schedulable.

CTT utilizes function called *CWork* to determine the cumulative work on a processor. These notions are formalized below:

```
BEGIN

 CWork(taskid) : RECURSIVE real =
     IF (1 > taskid)
        THEN 0
     ELSE IF (taskid = 1)
             THEN Execution(AnyTask(taskid)) × ceiling(t/Period(AnyTask(taskid)))
          ELSE

(Execution(AnyTask(taskid)) × ceiling(t/Period(AnyTask(taskid))) + CWork(taskid − 1))
          ENDIF
     ENDIF
       MEASURE (λ taskid: taskid)
   CTTest?(tid: task_id) : bool =
         (CWork(tid) = CWork(tid + 1)) ∧
          (CWork(tid) ≤ Period(AnyTask(tid)))
END
```

3.1.2 Formalization of Rate Monotonic Component

With constraints imposed on task attributes, we now formalize the priority assignment suggested by the rate monotonic algorithm. Essentially, a task with a lower period gets a higher priority *(RMA_priority)*. The complaince of this formalization to the rate monotonic algorithm is verified using the *RMA_OrderingTH* theorem, which has been proved trivially.

```
rate_monotonic: THEORY
 BEGIN
  IMPORTING task_model
  AnyTask: task_vector
```

```
TaskId1, TaskId2, TaskId3: VAR task_id

Number: posnat

task_priority: [task_id → real] =
     λ (TaskId1) : Number/Period(AnyTask(TaskId1))

RMA_priority: AXIOM
   (∀ TaskId1:
      ∀ TaskId2:
         Period(AnyTask(TaskId1)) > Period(AnyTask(TaskId2)) ⊃
           task_priority(TaskId1) < task_priority(TaskId2))

RMA_Ordering: AXIOM
   (∀ (TaskId1) :
      Period(AnyTask(TaskId1)) < Period(AnyTask(TaskId1 + 1)))

RMA_Orderingrev: AXIOM
   (∀ (TaskId1) :
      Number/Period(AnyTask(TaskId1)) >
        Number/Period(AnyTask(TaskId1 + 1)))

Processorload: [task_id → real] =
     λ (TaskId1) :
       (Execution(AnyTask(TaskId1))/Period(AnyTask(TaskId1)))

FeasilbilityCond: AXIOM
   (∀ (TaskId1) :
      2 ≥ expt((1 + Processorload(TaskId1)/Number), Number))

RMA_OrderingTH: THEOREM
   (∀ (TaskId1) :
      task_priority(TaskId1) > task_priority(TaskId1 + 1))

X, Y: nonneg_real

totalpriorityevents: [task_id → real] =
     λ (TaskId3) :
       ceiling((Y/Period(AnyTask(TaskId3))) − (X/Period(AnyTask(TaskId3))))

totalcomputedtime: [task_id → real] =
     λ (TaskId3) :
       totalpriorityevents(TaskId3) × Execution(AnyTask(TaskId3))

END rate_monotonic
```

3.1.3 Formalization of Rate Monotonic First Fit Component

This modules utilizes the task set generated by the rate monotonic component and using the CTT formulation from the task module defines a *AddTask* function. This function takes a task and assigns it on the first available processor on which it can be feasibly scheduled along with all other tasks that have already been assigned on that processor (First Fit rule). The function *TaskonProcessor* is formalized to infer whether a task exists on a processor or not. The function *Returnvalue* returns the total execution time of all the task instances. *NoPr* and *MinNoPr* denote the maximum and minimum number of processors, respectively. This notion is formalized below:

```
firstfit: THEORY
 BEGIN
  IMPORTING task_model
  TaskID: VAR task_id
  p: Processor
  AnyTask: task_vector
  t: posnat
  db: Processor_database
  FirstTask?(tid: task_id): bool = (tid = 0)
  TaskonProcessor?(tid: task_id, n: task_assignment, j: Processor): bool =

      TaskId(n) = tid ∧ (n ∈ db(j))
  Returnvalue(TaskID: task_id, n1: task_assignment, j1: Processor): real =
      IF (TaskonProcessor?(TaskID, n1, j1))
        THEN (Execution(AnyTask(TaskID)) × ceiling(t/Period(AnyTask(TaskID))))
      ELSE 0
      ENDIF
  j, k: VAR Processor
  NoPr: posnat = 10
  MinNoPr: posnat = 1
```

```
AddTask(n: task_assignment, ((k: posnat |
k ≥ MinNoPr ∧ k ≤ NoPr)),

 db: Processor_database):
 RECURSIVE Processor_database =
   IF FirstTask?(TaskId(n))
     THEN db WITH [(k) := (db(k) ∪ {n})]
   ELSE IF CTTest?(TaskId(n))
          THEN db WITH [(k) := (db(k) ∪ {n})]
        ELSE IF (NoPr − k = 0) THEN db
 ELSE AddTask(n, k+1, db) ENDIF
        ENDIF
   ENDIF
    MEASURE NoPr − k
END firstfit
```

3.1.4 Formalization of Primary Backup component

We formalize various requirements of primary backup approach in this component. We identify and specify various constraints imposed on the backup copies of a task. It is assumed that the execution time of backup task is less than or equal to that of primary task. Furthermore, primary and backup tasks are not assigned on the same processor. These are fomalized as PBAX1 and PBAX2 below.

Cumulative work done construct can be used to decide whether a backup copy should be *active* or *passive*. If the schedulability test for τ_i succeeds, then the worst-case execution time, ψ_i is less than T_i. If within each task's request period, the recovery time, $B_i = T_i - \psi_i$, is greater than its backup task's (β) execution time, i.e., $B_i \geq C_i'$ then β_i is scheduled as a passive copy; otherwise it is scheduled as an active copy.

We define functions *Pri, Act* and *Pas* that return a set of tasks that are of type primary, active and passive, respectively on a processor. We formalize these notions below.

```
PrimaryBackup: THEORY
 BEGIN

  IMPORTING rate_monotonic
  primary_task: TYPE = {n: task_assignment | TaskType(n) = primary}
  backup_task: TYPE = {n: task_assignment | TaskType(n) = backup}
  T: task_vector
  m: TaskInstance
  num: VAR task_assignment
  PR: VAR primary_task
  BK: VAR backup_task
  pro: AllocateProcessor
  db: Processor_database
  PBAX1: AXIOM
    ∀ (BK, (PR: primary_task | TaskId(PR) = TaskId(BK))):
      Execution(T(TaskId(BK))) ≤ Execution(T(TaskId(PR)))
  PBAX2: AXIOM
    ∀ (BK, (PR: primary_task | TaskId(PR) = TaskId(BK))):
      Period(T(TaskId(PR))) = Period(T(TaskId(BK)))
  PBAX3: AXIOM
    ∀ (BK, (PR: primary_task | TaskId(PR) = TaskId(BK))):
      pro(PR) ≠ pro(BK)
 get_status(n: task_assignment): backup_task_status =
        (IF TaskType(n) = backup ∧
(Period(T(TaskId(n))) – CWork(TaskId(n))) ≥ Period(T(TaskId(n)))
           THEN passive
         ELSE active
         ENDIF)


  Pri(j: Processor): set[task_assignment] =
      {n: task_assignment | (n ∈ db(j)) ∧ TaskType(n) = primary}
  Act(j: Processor): set[task_assignment] =
      {n: task_assignment | (n ∈ db(j)) ∧ TaskStatus(n) = active}
  pas(j: Processor): set[task_assignment] =
      {n: task_assignment | (n ∈ db(j)) ∧ TaskStatus(n) = passive}
```

```
Active_Recover (j, f: Processor) : set [task_assignment] =
    {n: task_assignment | (n ∈ Act(j)) ∧ (n ∈ Pri(f))}
Passive_Recover (j, f: Processor) : set [task_assignment] =
    {n: task_assignment | (n ∈ pas(j)) ∧ (n ∈ Pri(f))}
Rec (j, f: Processor) : set [task_assignment] =
    (Active_Recover(j, f) ∪ Passive_Recover(j, f))
NonFaultyGrp (j: Processor) : set [task_assignment] =
    {assign: task_assignment | (assign ∈ (Pri(j) ∪ Act(j)))}
OneFaultyGrp (j, f: Processor) : set [task_assignment] =
    {assign1: task_assignment | (assign1 ∈ (Pri(j) ∪ Rec(j, f)))}
PBTH: THEOREM
  ∀ ((num: task_assignment | TaskType (num) = primary)) :
    TaskType (num) ≠ backup
END PrimaryBackup
```

This formalization is verified using the *PBTH* theorem, which verifies that the primary and backup copies of a task are not assigned to the same processor.

So far, we have discussed our formal framework, and have presented formal theories of basic components. These formal constructs are extensively used in assigning primary and backup tasks on different processors using FTRMFF protocol. The next section discusses utilization of these formal theories to specify the FTRMFF protocol.

3.2 Putting Components Together to Formalize FTRMFF

The building blocks specified earlier can be used to specify the FTRMFF protocol. This process is done by importing the formal theories in the FTRMFF protocol specification in PVS. In this section we explain the FTRMFF protocol specification.

3.2.1 Verifi cation and Analysis of FTRMFF Protocol

The *task, rate montonic, RMFF* and *primary backup* theories are imported and utilized in the FTRMFF specification. The fault-tolerant version of *CTT* which was specified in the task model, requires two sets of tasks to evaluate schedulability criteria under no-fault and one-fault conditions. We characterize the *CTT* under two two task failure scenarios. In order to achieve this, we need to define two group of tasks, namely *NonFaultyGrp* and *OneFaultyGrp*. *NonfaultyGrp* denotes a union of a task to be assigned on a processor, and primary and active backup tasks on that processor.

Similarly, *OneFaultyGrp* denotes a union of a task to be assigned on a processor and primary task on that processor and a union of passive and active recovery tasks (*Rec*) defined earlier. They are formally specified in the Primary Backup Module as follows:

$$
\begin{aligned}
&\text{Pri}(j:\ \text{Processor}):\ \text{set[task\_assignment]}\ =\\
&\quad \{n:\ \text{task\_assignment}\ \mid\ (n \in \text{db}(j))\ \wedge\ \text{TaskType}(n)\ =\ \text{primary}\}\\
&\text{Act}(j:\ \text{Processor}):\ \text{set[task\_assignment]}\ =\\
&\quad \{n:\ \text{task\_assignment}\ \mid\ (n \in \text{db}(j))\ \wedge\ \text{TaskStatus}(n)\ =\ \text{active}\}\\
&\text{pas}(j:\ \text{Processor}):\ \text{set[task\_assignment]}\ =\\
&\quad \{n:\ \text{task\_assignment}\ \mid\ (n \in \text{db}(j))\ \wedge\ \text{TaskStatus}(n)\ =\ \text{passive}\}\\
&\text{Active\_Recover}(j,\ f:\ \text{Processor}):\ \text{set[task\_assignment]}\ =\\
&\quad \{n:\ \text{task\_assignment}\ \mid\ (n \in \text{Act}(j))\ \wedge\ (n \in \text{Pri}(f))\}\\
&\text{Passive\_Recover}(j,\ f:\ \text{Processor}):\ \text{set[task\_assignment]}\ =\\
&\quad \{n:\ \text{task\_assignment}\ \mid\ (n \in \text{pas}(j))\ \wedge\ (n \in \text{Pri}(f))\}
\end{aligned}
$$

```
Rec (j, f: Processor) : set [task_assignment] =
    (Active_Recover(j, f) ∪ Passive_Recover(j, f))
NonFaultyGrp (j: Processor) : set [task_assignment] =
    {assign: task_assignment | (assign ∈ (Pri(j) ∪ Act(j)))}
OneFaultyGrp (j, f: Processor) : set [task_assignment] =
    {assign1: task_assignment | (assign1 ∈ (Pri(j) ∪ Rec(j, f)))}
```

3.2.1.1 Formalization of Schedulability Criteria

The primary and backup tasks should satisfy a certain criteria in order to be scheduled. The following denotes the schedulability criteria of the task set: [Bert1999]

Let the sets *primary(P_j)* and *backup(P_j)* represent the primary and backup copies assigned to processor P_j. Let *recover(P_j, P_f)* represents the union of two sets, namely *activeRecover(P_j, P_f)* and *passiveRecover(P_j, P_f)*, where these sets consist of the active and passive copies, respectively, assigned to P_j such that their primary copies are assigned to P_f. The function $\phi(k,t)$ gives the overall number of requests of a backup copy β_k during $[0,t]$.

o Let $\sigma = primary(P_j) \cup active(P_j)$ be the set of periodic tasks given in priority order which are assigned to processor P_j. All the periodic requests of tasks in σ will meet the deadlines iff

$$\max_{\tau_k,\beta_k \in \sigma} \min_{0<t\leq T_k} \left\{ \sum_{\tau_k \in \sigma} C_k \lceil t/T_k \rceil /t + \sum_{\beta_k \in \sigma} C'_k \phi(k,t)/t \right\} \leq 1$$

o Consider any processor P_j, and let at time θ a failure be detected in processor P_f. Let $\sigma = primary(P_j) \cup recover(P_j, P_f)$ be the set of periodic tasks given in priority order which are assigned to processor P_j. All the periodic requests of tasks in σ will meet the deadlines for any θ iff

$$\max_{\tau_k, \beta_k \in \sigma} \min_{0<t\leq V_k} \left\{ \sum_{\tau_k \in \sigma} C_k \lceil t/T_k \rceil /t + \sum_{\beta_k \in \sigma} C'_k \phi(k,t)/t \right\} \leq 1$$

where V_k is equal to T_k for a primary copy or an active backup copy and to B_k (recovery time for the backup copy β_k) for a passive backup copy.

In order to extend the *completion time test* for fault-tolerant version, we consider a task set which contains both primary and backup tasks which must be scheduled all together on a single processor. To formalize schedulability guarantees for both primary and backup tasks, we first specify the notion of the overall number of requests of a backup copy during [0,t], $\phi(k,t)$: [Bert1999]

$$\phi(k,t) = \begin{cases} \lceil t/T_k \rceil & \text{if } \beta_k \text{ is active} \\ 1 & \text{if } \beta_k \text{ is passive and } t \leq B_k \\ 1 + \lceil (t - B_k)/T_k \rceil & \text{if } \beta_k \text{ is passive and } t > B_k \end{cases}$$

Here, B_k is the recovery time of a task. We formally represent $\phi(k,t)$ as follows.

```
Time_to_recover(taskid: task_id) : real =
      Period(AnyTask(taskid)) − CWork(taskid)
 No_Of_backup_requests(taskid: task_id,

 ((num: task_assignment | TaskId(num) = taskid)),time: posreal) posnat =
      IF TaskStatus(num) = active
         THEN ceiling(time/Period(AnyTask(TaskId(num))))
      ELSE IF TaskStatus(num) = passive
```

```
            THEN  (IF time > Time_to_recover(TaskId(num))
                THEN
1 + abs(ceiling((time − Time_to_recover(TaskId(num)))/
Period(AnyTask(TaskId(num)))))
                            ELSE 1
                            ENDIF)
              ELSE 1
              ENDIF
        ENDIF
```

Essentially, based on the insights gained in *first fit* theory to calculate cumulative work done, we calculate the work done by both primary and backup copies scheduled on a processor over a time interval $[0, t]$. Below, $W1$ is the work done by a primary and $W2$ is that by backup copies.

```
 TaskiD: task_id
  $t_1$, $t_2$: VAR nat
  ta, tb: VAR nat
  $n$: task_assignment
  $W_1$($t_1$, $t_2$) : RECURSIVE real =
    IF ($t_1$ > $t_2$)
      THEN 0
    ELSE IF ($t_2$ = $t_1$)
            THEN Execution(AnyTask(TaskiD))
          ELSE

 (Execution(AnyTask(TaskiD)) × ceiling($t_2$/Period(AnyTask(TaskiD))) +
 W1($t_1$, $t_2 - 1$))
          ENDIF
    ENDIF
     MEASURE ($t_2$)
  $W_2$(ta, tb) : RECURSIVE real =
    IF (ta > tb)
      THEN 0
    ELSE IF (tb = ta)
```

```
          THEN  Execution (AnyTask (TaskiD) )
        ELSE

(Execution(AnyTask(TaskiD)) × No_Of_backup_requests(TaskId(n),  n,  tb) +
W2(ta,  tb − 1))
        ENDIF
   ENDIF
    MEASURE  (tb)
  TotalWork_Pri_bac (t1,  t2) :  real  =  W1(t1,  t2) + W2(t1,  t2)
```

Now, we formalize CTTs for no-fault and one-fault cases. For these CTTs, we utilize Boolean expressions *FTRMFF_non_faulty?* and *FTRMFF_one_fault?* that basically check for schedulability of tasks on a processor belonging to non-faulty group and one-faulty group, respectively. The *NoFaultCTT?* and *OneFaultCTT?* boolean expressions determine the CTT for non faulty group and one faulty group processors respectively.

```
time0,  time1:  nat
  ins:  TaskInstance
  FTRMFF_non_faulty? (j:  Processor) :  bool  =
        (∀  ( (number:  task_assignment  |  (number ∈ NonFaultyGrp(j))) ) :
           TotalWork_Pri_bac (time0,  time1)  ≤  1)
  NoFaultCTT? (n:  task_assignment,  j:  Processor) :  bool  =
        NonFaultyGrp (j)  =  (singleton(n) ∪ NonFaultyGrp(j))  ⊃
         FTRMFF_non_faulty? (j)
  FTRMFF_one_fault? (j,  f:  Processor) :  bool  =
        (∀  ( (n:  task_assignment  |  (n ∈ OneFaultyGrp(j,  f))) ) :
           TotalWork_Pri_bac (time0,  time1)  ≤  1)
  OneFaultCTT? (n:  task_assignment,  j,  f:  Processor) :  bool  =
        OneFaultyGrp (j,  f)  =  (singleton(n) ∪ OneFaultyGrp(j,  f))  ⊃
         FTRMFF_one_fault? (j,  f)
```

In order to assign a particular task under FTRMFF policy, we have divided the assignment process into three functions that for each primary, active and passive backup. We make our decision on an assignment based on three boolean functions, namely *PrimaryADD*, *ActiveBacAdd*, and *PassiveBacAdd* that in turn utilize *NoFaultCTT* and *OneFaultCTT*.

```
f: Processor
 NoPr: TYPE+ = posnat
 MinNoPr: TYPE+ = posnat
 Pr_AX: AXIOM NoPr > MinNoPr
 PrimaryADD?(n: task_assignment, k: Processor): bool =
     NoFaultCTT?(n, k) ∧ OneFaultCTT?(n, k, f)
 add_pri(n: task_assignment, ((k: posnat | k ≥ MinNoPr ∧ k ≤ NoPr)),
  db: Processor_database):
 RECURSIVE Processor_database =
   IF PrimaryADD?(n, k)
     THEN db WITH [(k) := (db(k) ∪ {n})]
   ELSE IF (NoPr − k = 0) THEN db ELSE add_pri(n, k+1, db) ENDIF
   ENDIF
    MEASURE NoPr − k
 ActiveBacAdd?(bk: backup_task, k: Processor): bool =
     NoFaultCTT?(bk, k) ∧ OneFaultCTT?(bk, k, pro(bk))
 add_act(((n: task_assignment | TaskType(n)


 = backup)), ((k: posnat | k ≥ MinNoPr ∧ k ≤ NoPr)),
          db: Processor_database):
 RECURSIVE Processor_database =
   IF ActiveBacAdd?(n, k)
     THEN db WITH [(k) := (db(k) ∪ {n})]
   ELSE IF (NoPr − k = 0) THEN db ELSE add_act(n, k+1, db) ENDIF
   ENDIF
    MEASURE NoPr − k
 PassiveBacAdd?(n: task_assignment, k: Processor): bool =
     OneFaultCTT?(n, k, pro(n))
```

```
add_pas(n: task_assignment, ((k: posnat | k ≥ MinNoPr ∧ k ≤ NoPr)),
 db: Processor_database):
RECURSIVE Processor_database =
  IF PassiveBacAdd?(n, k)
    THEN db WITH [(k) := (db(k) ∪ {n})]
  ELSE IF (NoPr − k = 0) THEN db ELSE add_pas(n, k+1, db) ENDIF
  ENDIF
   MEASURE NoPr − k

db: Processor_database

j, k: VAR Processor
```

To add a set of tasks including both primary and backup, we utilize the function *AddTaskN* which adds these tasks to the processor database. This function in turn calls *add_pri, add_act* and *add_pas* for each task type as discussed earlier.

```
AddTaskN(n: task_assignment, ((k: posnat | k ≥ MinNoPr ∧ k ≤ NoPr)),
db: Processor_database):
Processor_database =IF TaskType(n) = primary
      THEN add_pri(n, k, db)
    ELSE IF TaskStatus(n) = active
    THEN add_act(n, k, db) ELSE
    add_pas(n, k, db)

ENDIF
    ENDIF
Anytask: task_vector
```

3.2.1.2 Verifi cation of the FTRMFF specifi cation

The different theories of allocation and scheduling to build up FTRMFF allocation and scheduling policies are verified by theorems formulated in the FTRMFF module. The following properties are verified:

1) The basic functionality which the tasks should exhibit is to be arranged rate monotonically in each of the processor. The following theorem verifies this property by allocating tasks to processors.

$$
\begin{array}{l}
\text{TestDb1}: \ \text{THEOREM} \\
\quad \forall \ (n\text{: task\_assignment}, \ ((k\text{: Processor} \mid k \geq \text{MinNoPr} \\
\quad \wedge \ k < \text{NoPr} - 1)), \\
\quad (\text{db: Processor\_database} \mid \text{db} = \text{AddTaskN}(n, \ k, \ \text{db})), \ \text{TaskId: task\_id}) : \\
\qquad (\text{TaskId}(n) = \text{b2n}((n \in \text{db}(k)))) \supset \\
\qquad\quad (\text{task\_priority}(\text{TaskId}) > \text{task\_priority}(\text{TaskId} + 1))
\end{array}
$$

2) Fault tolerance is an important property of the FTRMFF protocol. This is modeled by ensuring that the backup task is active when the primary task has failed. The following theorem verifies this property.

$$
\begin{array}{l}
\text{TestDb2}: \ \text{THEOREM} \\
\quad \forall \ (n\text{: task\_assignment}, \ ((k\text{: Processor} \mid k \geq \text{MinNoPr} \\
\quad \wedge \ k < \text{NoPr} - 1)), \\
\quad (\text{db: Processor\_database} \mid \text{db} = \text{AddTaskN}(n, \ k, \ \text{db})), \ \text{TaskId: task\_id}) : \\
\qquad (\text{TaskId}(n) = \text{b2n}((n \in \text{db}(k))) \ \wedge \\
\qquad (\text{TaskType}(n) \neq \text{primary}) \supset \\
\qquad\quad (\text{TaskType}(n) = \text{backup} \wedge \ \text{TaskStatus}(n) = \text{active})
\end{array}
$$

3) A Backup task can exist in either passive or active state. Whenever a fault occurs the backup changes its state from passive to active and then starts running. The following theorem verifies that the FTRMFF specification does not allow any passive backup to execute.

TestDb3 : THEOREM

$\forall$ (n: task_assignment, ((k: Processor | $k \geq$ MinNoPr $\wedge$ $k <$ NoPr $- 1$)),

(db: Processor_database | db = AddTaskN(n, k, db)), TaskId: task_id) :

(TaskId(n) = b2n(($n \in$ db(k))) $\wedge$

(TaskStatus(n) $\neq$ active) $\supset$

(TaskType(n) = backup $\wedge$ TaskStatus(n) = passive)

3.2.1.3 Verifi cation of FTRMFF: Results

The three theorems discussed earlier utilize the previously defined theories inorder to be proven. The three theorems were proven successfully and the proof summary of the mechanized proving is as follows:

```
Proof summary for theory FTRMFF
   No_Of_backup_requests_TCC1....proved - complete   [shostak](0.49 s)
   W1_TCC1.......................proved - complete   [shostak](0.19 s)
   W1_TCC2.......................proved - complete   [shostak](0.03 s)
   W2_TCC1.......................proved - complete   [shostak](0.12 s)
   add_pri_TCC1..................proved - complete   [shostak](0.28 s)
   add_pri_TCC2..................proved - complete   [shostak](0.73 s)
   add_pri_TCC3..................proved - complete   [shostak](0.22 s)
   add_act_TCC1..................proved - complete   [shostak](0.75 s)
   add_act_TCC2..................proved - complete   [shostak](0.23 s)
   add_pas_TCC1..................proved - complete   [shostak](0.87 s)
   add_pas_TCC2..................proved - complete   [shostak](0.20 s)
   AddTaskN_TCC1.................proved - complete   [shostak](0.16 s)
   TestDb1_TCC1..................proved - complete   [shostak](0.20 s)
```

```
    TestDb1......................proved - complete   [shostak](3.01 s)
    TestDb2......................proved - complete   [shostak](0.93 s)
    TestDb3......................proved - complete   [shostak](0.88 s)
    Theory totals: 16 formulas, 16 attempted, 16 succeeded (9.29 s)
Grand Totals: 33 proofs, 33 attempted, 33 succeeded (16.80 s)
```

The mechanized proof of TestDb1 in the PVS tool is illustrated in Section A.1.

So far, we have demonstrated the specification of the building blocks of the FTRMFF protocol as well as the analysis and verification of the composed theory from these building blocks. We however would like to re-emphasize that the primary objective of this thesis is to demonstrate the reusability of the formal constructs to specify related protocols. Therefore, we utilize the components to specify and verify a new protocol , namely the Fault Tolerant Rate Monotonic Next Fit Protocol(FTRMNF).

3.3 Specification and Verification of FTRMNF protocol

FTRMNF protocol essentially works on the similar lines as FTRMFF protocol, with the difference lying only in the allocation schema. FTRMFF utilizes the Rate Monotonic First Fit (RMFF) algorithm to allocate the tasks to processors, while FTRMNF utilizes the Rate Monotonic Next Fit (RMNF) algorithm to allocate the tasks to processors.

3.3.1 Formalization of the components of FTRMNF

The components for the FTRMNF protocol are the *task model*, *rate monotonic*, *rate monotonic next fit* and *primary backup*. All of these components(except for rate monotonic next fit) are essentially the same components which form the FTRMFF protocol specification.

The previously defined specifications can be reused and composed together to form the FTRMNF protocol.

We discuss the new theory that has been developed in regard to the FTRMNF component formalization.

3.3.1.1 Rate Monotonic Next Fit Component

This modules utilizes the task set generated by the rate monotonic component and using the CTT formulation from the task module defines a *AddTask* function. This function takes a task and assigns it on the next available free processor on which it can be feasibly scheduled along with all other tasks that have already been assigned on that processor (Next Fit rule). *NoPr* and *MinNoPr* denote the maximum and minimum number of processors, respectively. This notion is formalized below:

```
nextfit: THEORY
 BEGIN

  IMPORTING task_model

  TaskID: VAR task_id
  p: Processor
  AnyTask: task_vector
  t: posnat
  db: Processor_database
  FirstTask?(tid: task_id) : bool = (tid = 0)
  TaskonProcessor?(tid: task_id, n: task_assignment, j: Processor) : bool =
      TaskId(n) = tid ∧ (n ∈ db(j))
  Returnvalue(TaskID: task_id, n1: task_assignment, j1: Processor) : real =
      IF (TaskonProcessor?(TaskID, n1, j1))
        THEN (Execution(AnyTask(TaskID)) × ceiling(t/Period(AnyTask(TaskID))))
      ELSE 0
      ENDIF
  j, k: VAR Processor
  NoPr: posnat = 10
  MinNoPr: posnat = 1
  PresentK: VAR posnat
  AddTask(n: task_assignment, PresentK,
((k: posnat | (k ≥ MinNoPr ∧ k ≤ NoPr ∧ k = PresentK))),
          db: Processor_database) :
  RECURSIVE Processor_database =
    IF FirstTask?(TaskId(n))
      THEN db WITH [(k) := (db(k) ∪ {n})]
    ELSE IF CTTest?(TaskId(n))
           THEN db WITH [(k) := (db(k) ∪ {n})]
         ELSE IF (NoPr − k
 = 0) THEN db ELSE
 AddTask(n, k+1, k+1,
 db)

ENDIF
         ENDIF
    ENDIF
     MEASURE NoPr − k
 END nextfit
```

3.4 Putting Components Together to Formalize FTRMNF

The building blocks specified earlier can be used to specify the FTRMNF protocol. This process is done by importing the formal theories in the FTRMNF protocol specification in PVS. In this section we explain the FTRMNF protocol specification.

3.4.1 Verifi cation and Analysis of FTRMNF protocol

The *task* , *rate montonic, RMNF* and *primary backup* theories are imported and utilized in the FTRMNF specification. We emphasize that the FTRMFF and FTRMNF protocols differ only in the allocation methods, the other strategies are essentially the same.

3.4.1.1 Formalization of Schedulability Criteria

The primary and backup tasks should satisfy a certain criteria in order to be scheduled. The following denotes the schedulability criteria of the task set:

Let the sets *primary(P_j)* and *backup(P_j)* represent the primary and backup copies assigned to processor P_j. Let *recover(P_j, P_f)* represents the union of two sets, namely *activeRecover(P_j, P_f)* and *passiveRecover(P_j, P_f)*, where these sets consist of the active and passive copies, respectively, assigned to P_j such that their primary copies are assigned to P_f. The function $\phi(k, t)$ gives the overall number of requests of a backup copy β_k during $[0, t]$.

o Let $\sigma = primary(P_j) \cup active(P_j)$ be the set of periodic tasks given in priority order which are assigned to processor P_j. All the periodic requests of tasks in σ will meet the deadlines iff

$$\max_{\tau_k, \beta_k \in \sigma} \min_{0<t\leq T_k} \left\{ \sum_{\tau_k \in \sigma} C_k \lceil t/T_k \rceil /t + \sum_{\beta_k \in \sigma} C'_k \phi(k,t)/t \right\} \leq 1$$

o Consider any processor P_j, and let at time θ a failure be detected in processor P_f. Let $\sigma = primary(P_j) \cup recover(P_j, P_f)$ be the set of periodic tasks given in priority order which are assigned to processor P_j. All the periodic requests of tasks in σ will meet the deadlines for any θ iff

$$\max_{\tau_k,\beta_k\in\sigma}\ \min_{0<t\leq V_k}\left\{\sum_{\tau_k\in\sigma} C_k\lceil t/T_k\rceil/t + \sum_{\beta_k\in\sigma} C'_k\phi(k,t)/t\right\}\leq 1$$

where V_k is equal to T_k for a primary copy or an active backup copy and to B_k (recovery time for the backup copy β_k) for a passive backup copy.

In order to extend the *completion time test* for fault-tolerant version, we consider a task set which contains both primary and backup tasks which must be scheduled all together on a single processor. To formalize schedulability guarantees for both primary and backup tasks, we first specify the notion of the overall number of requests of a backup copy during [0,t], $\phi(k,t)$:

$$\phi(k,t) = \begin{cases}\lceil t/T_k\rceil & \text{if } \beta_k \text{ is active}\\ 1 & \text{if } \beta_k \text{ is passive and t} \leq \mathrm{B_k}\\ 1+\lceil (t-B_k)/T_k\rceil & \text{if } \beta_k \text{ is passive and t} > \mathrm{B_k}\end{cases}$$

Here, B_k is the recovery time of a task. We formally represent $\phi(k,t)$ as follows.

```
Time_to_recover(taskid: task_id) : real =
      Period(AnyTask(taskid)) − CWork(taskid)
 No_Of_backup_requests(taskid: task_id,
 ((num: task_assignment | TaskId(num) = taskid)),time: posreal):posnat =
      IF TaskStatus(num) = active
         THEN ceiling(time/Period(AnyTask(TaskId(num))))
      ELSE IF TaskStatus(num) = passive
      THEN
```

```
            (IF  time  >  Time_to_recover(TaskId(num))
          THEN

      1 + abs(ceiling((time − Time_to_recover(TaskId(num)))/
      Period(AnyTask(TaskId(num)))))
          ELSE  1
                          ENDIF)
            ELSE  1
            ENDIF
        ENDIF
```

Essentially, based on the insights gained in *next fit* theory to calculate cumulative work done, we calculate the work done by both primary and backup copies scheduled on a processor over a time interval $[0, t]$. Below, $W1$ is the work done by a primary and $W2$ is that of by backup copies.

```
 TaskiD:  task_id
  $t_1$,  $t_2$:  VAR  nat
  ta,  tb:  VAR  nat
  $n$:  task_assignment
  $W_1(t_1, t_2)$ :  RECURSIVE  real  =
    IF  $(t_1 > t_2)$
      THEN  0
    ELSE  IF  $(t_2 = t_1)$
              THEN  Execution(AnyTask(TaskiD))
          ELSE

(Execution(AnyTask(TaskiD)) × ceiling($t_2$/Period(AnyTask(TaskiD)))) + $W_1(t_1, t_2 - 1)$)
          ENDIF
    ENDIF
     MEASURE  $(t_2)$
  $W_2$(ta,  tb) :  RECURSIVE  real  =
    IF  (ta > tb)
```

```
        THEN  0
    ELSE  IF  (tb = ta)
              THEN
    Execution (AnyTask (TaskiD) )
            ELSE

(Execution(AnyTask(TaskiD)) × No_Of_backup_requests(TaskId(n),  n,  tb) +
W2(ta,  tb − 1))
            ENDIF
    ENDIF
      MEASURE  (tb)
  TotalWork_Pri_bac (t1,  t2) :  real  =  W1(t1,  t2) + W2(t1,  t2)
```

Now, we formalize CTTs for no-fault and one-fault cases. For these CTTs, we utilize Boolean expressions *FTRMNF_non_faulty?* and *FTRMNF_one_fault?* that basically check for schedulability of tasks on a processor belonging to non-faulty group and one-faulty group, respectively.

```
 time0,  time1:  nat
  ins:  TaskInstance
  FTRMNF_non_faulty? (j:  Processor) :  bool  =
        (∀  ( (number:  task_assignment  |  (number ∈ NonFaultyGrp(j))) ) :
            TotalWork_Pri_bac (time0,  time1)  ≤  1)
  NoFaultCTT? (n:  task_assignment,  j:  Processor) :  bool  =
        NonFaultyGrp (j)  =  (singleton(n) ∪ NonFaultyGrp(j))  ⊃
          FTRMNF_non_faulty? (j)
  FTRMNF_one_fault? (j,  f:  Processor) :  bool  =
        (∀  ( (n:  task_assignment  |  (n ∈ OneFaultyGrp(j,  f))) ) :
            TotalWork_Pri_bac (time0,  time1)  ≤  1)
  OneFaultCTT? (n:  task_assignment,  j,  f:  Processor) :  bool  =
        OneFaultyGrp (j,  f)  =  (singleton(n) ∪ OneFaultyGrp(j,  f))  ⊃
          FTRMNF_one_fault? (j,  f)
```

In order to assign a particular task under FTRMNF policy, we have divided the assignment process into three functions that for each primary, active and passive backup. We make our decision on an assignment based on three boolean functions, namely *PrimaryADD*, *ActiveBacAdd*, and *PassiveBacAdd* that inturn utilize *NoFaultCTT* and *OneFaultCTT*.

```
f: Processor
 NoPr: TYPE+ = posnat
 MinNoPr: TYPE+ = posnat
 Pr_AX: AXIOM NoPr > MinNoPr
 PrimaryADD?(n: task_assignment, k: Processor) : bool =
     NoFaultCTT?(n, k) ∧ OneFaultCTT?(n, k, f)
 add_pri(n:task_assignment, ((k: posnat|k ≥ MinNoPr ∧ k ≤ NoPr)), db:
 Processor_database) :
 RECURSIVE Processor_database =
   IF PrimaryADD?(n, k)
     THEN db WITH [(k) := (db(k) ∪ {n})]
   ELSE IF (NoPr − k = 0) THEN db ELSE add_pri(n, k+1, db) ENDIF
   ENDIF
    MEASURE NoPr − k
 ActiveBacAdd?(bk: backup_task, k: Processor) : bool =
     NoFaultCTT?(bk, k) ∧ OneFaultCTT?(bk, k, pro(bk))
 add_act(((n: task_assignment | TaskType(n)
 = backup)),

((k: posnat | k ≥ MinNoPr ∧ k ≤ NoPr)),
           db: Processor_database) :
 RECURSIVE Processor_database =
   IF ActiveBacAdd?(n, k)
     THEN db WITH [(k) := (db(k) ∪ {n})]
   ELSE IF (NoPr − k = 0) THEN db ELSE add_act(n, k+1, db) ENDIF
   ENDIF
    MEASURE NoPr − k
 PassiveBacAdd?(n: task_assignment, k: Processor) : bool =
     OneFaultCTT?(n, k, pro(n))
```

```
add_pas(n:task_assignment, ((k: posnat | k ≥ MinNoPr ∧ k ≤ NoPr)),db:
Processor_database):
RECURSIVE Processor_database =
  IF PassiveBacAdd?(n, k)
    THEN db WITH [(k) := (db(k) ∪ {n})]
  ELSE IF (NoPr − k = 0) THEN db ELSE add_pas(n, k+1, db) ENDIF
  ENDIF
   MEASURE NoPr − k
db: Processor_database
j, k: VAR Processor
```

To add a set of tasks including both primary and backup, we utilize the function *AddTaskN* which adds these tasks to the processor database. This function inturn calls *add_pri, add_act* and *add_pas* for each task type as discussed earlier.

```
AddTaskN(n: task_assignment, ((k: posnat | k ≥ MinNoPr ∧ k ≤ NoPr)),
 db: Processor_database):
Processor_database =
    IF TaskType(n) = primary
      THEN add_pri(n, k, db)
    ELSE IF TaskStatus(n) = active


    THEN add_act(n, k, db) ELSE add_pas(n, k, db) ENDIF
    ENDIF
Anytask: task_vector
```

3.4.1.2 Verifi cation of the FTRMNF specifi cation

The using of the different theories of allocation and scheduling to build up FTRMNF allocation and scheduling policies are verified by theorems formulated in the FTRMNF module. The following properties are verified:

1) The tasks added to the processors in the database should follow the rate monotonic

principle. The following theorem verifies this property.

TestDb1 : THEOREM

$\forall$ (n : task_assignment, ((k : Processor | $k \geq$ MinNoPr$\wedge$ k < NoPr − 1)) ,

(db : Processor_database | db = AddTaskN (n, k, db)) , TaskId : task_id) :

(TaskId (n) = b2n (($n \in$ db(k)))) $\supset$

(task_priority (TaskId) > task_priority (TaskId + 1))

2) To provide fault tolerance a faulty processor running the primary task should result in the backup task getting active. The following theorem verifies this property in the FTRMNF specification.

TestDb2 : THEOREM

$\forall$ (n : task_assignment, ((k : Processor | $k \geq$ MinNoPr

$\wedge$ k < NoPr − 1)) ,

(db : Processor_database | db = AddTaskN (n, k, db)) , TaskId : task_id) :

(TaskId (n) = b2n (($n \in$ db(k))) $\wedge$

(TaskType (n) $\neq$ primary) $\supset$

(TaskType (n) = backup$\wedge$ TaskStatus (n) = active)

3) The backup should be in a running state only when it is active. Passive backup cannot execute. This theorem verifies this property.

TestDb3 : THEOREM

$\forall$ (n : task_assignment, ((k : Processor | $k \geq$

MinNoPr $\wedge$ k < NoPr − 1)) ,

(db : Processor_database | db = AddTaskN (n, k, db)) , TaskId : task_id) :

(TaskId (n) = b2n (($n \in$ db(k))) $\wedge$

$$(\text{TaskStatus}(n) \neq \text{active}) \supset$$
$$(\text{TaskType}(n) = \text{backup} \land \text{TaskStatus}(n) = \text{passive})$$

3.4.1.3 Verifi cation of FTRMNF: Results

The three theorems discussed earlier utilize the previously defined theories in order to be proven. The three theorems were proven successfully and the proof summary in the mechanized proving using PVS is illustrated below:

```
 Proof summary for theory FTRMNF
    No_Of_backup_requests_TCC1....proved - complete(0.49s)
    W1_TCC1......................proved - complete(0.19s)
    W1_TCC2......................proved - complete(0.03s)
    W2_TCC1......................proved - complete(0.12s)
    add_pri_TCC1.................proved - complete(0.28s)
    add_pri_TCC2.................proved - complete(0.73s)
    add_pri_TCC3.................proved - complete(0.22s)
    add_act_TCC1.................proved - complete(0.75s)
    add_act_TCC2.................proved - complete(0.23s)
    add_pas_TCC1.................proved - complete(0.87s)
    add_pas_TCC2.................proved - complete(0.20s)
    AddTaskN_TCC1................proved - complete(0.16s)
    TestDb1_TCC1.................proved - complete(0.20s)
    TestDb1......................proved - complete(3.01s)
    TestDb2......................proved - complete(0.93s)
    TestDb3......................proved - complete(0.88s)
Theory totals: 16 formulas, 16 attempted, 16 succeeded (9.29 s)
Grand Totals: 33 proofs, 33 attempted, 33 succeeded (16.80 s)
```

We have demonstrated that the reusing of the previously specified building blocks for FTRMFF can be reused to specify a related protocol FTRMNF. The main objective was to show the reduced effort in formalization process. The FTRMNF protocol belongs to the same category (dependable periodic task scheduling and allocation) as the FTRMFF. We next demonstrate that a development of a library can minimise the effort in analyzing other scheduling protocols too. In the next chapter, we discuss a dependable aperiodic task scheduling and allocation protocol, and demonstate how the theories that have been developed so far can be reused to reduce the effort in the analysis and verification.

Chapter 4

Formal Analysis of Dependable Aperiodic Task Scheduling and Allocation Protocol

We emphasize on the concept of reuse to reduce the effort in protocol analysis and verification. The earlier chapter demonstrated the concept of reuse by utilizing the building blocks of the FTRMFF protocol to specify and verify the FTRMNF protocol. In this chapter we first introduce the dependable aperiodic scheduling and allocation protocol as described in [Ghos1997b]. We then identify the building blocks of the protocol and finally reuse the previously defined components to specify and verify the protocol.

4.1 Dependable Aperiodic Task Scheduling and Allocation Protocol

Due to the critical nature of tasks in a hard real-time system, it is essential that every task admitted in the system completes its execution even in the presence of faults. The scheduling protocol presented in [Ghos1997b] aims to achieve that. The protocol uses the primary backup approach where the backup task activates whenever the primary task fails.

The protocol proposes to schedule multiple copies of dynamic, aperiodic, nonpreemptive tasks in the system, and uses two techniques called deallocation and overloading to achieve high acceptance ratio which is the percentage of arriving tasks scheduled by the system.

4.1.1 Building Blocks of the Aperiodic Task Scheduling and Allocation Protocol

In Chapter 2 we had discussed the various building blocks present in all dependable scheduling protocols for multiprocessor systems. We describe the different building blocks with respect to the aperiodic task scheduling algorithm.

1. *System model*: We consider a multiprocessor network of processors connected via a shared memory. The tasks are independent in nature so there are no dependency and precedence rules considered.

2. *Failure model:* We limit our present failure modeling of the scheduling protocol to fail stop condition only.

3. *Scheduling* : We use a simple slack based dynamic scheduling algorithm in the implementation for aperiodic tasks done by us in this thesis. The slack is present to tolerate faults as well as ensure a high acceptance ratio.

4. *Resource Allocation*: Resource allocation to a task is the most critical task in any aperiodic task model. To ensure that the resource is allocated to the maximum number of tasks that arrive two methods have been proposed:

   (a) *Backup overloading which* involves scheduling of more than one backup in the same time slot on a processor.

   (b) *Backup deallocation* which reclaims the resource which was reserved by the backup task once the corresponding primary task completes it execution.

5. *Fault tolerance:* The fault tolerance in the scheduling and allocation protocol involves the primary backup approach.

After describing the various building blocks for the fault tolerant real time scheduling protocols we discuss the specification of these components and verification of the protocol in PVS.

4.2 Formalization of the Components of the Dependable Aperiodic Task Scheduling and Allocation Protocol

We formally specify the components of the aperiodic task protocol in PVS. The specification of the components characterizes the working of the components in an abstract manner. We use theorem proving to ascertain the correctness of theories in each of the components. It should however be noted that our emphasis lies on reuse of the previously developed theories in the specification of this protocol.

We identify two components which require to be modeled

4.2.1 Formalization of Task Model

We begin with formally specifying the task model that has been used in the development of formal specification. At first we define types that are going to be used in formally specifying the attributes of a task set. We define *task_property* as a record type containing task arrival time, task ready time, task start time, Task end time, deadline, maximum computation time, task window, execution time and phase. This record is assigned to a task ID resulting in a *task_vector* type. We then declare a type to tag a task either as a primary or a backup as well as characterize a backup task as passive or active. Based on these primitives, we define a record type *task_assignment* which for each task ID associates its type (primary or backup) and status as active or passive.

```
task_model: THEORY
 BEGIN
  task_id: TYPE+ = posnat
```

```
TBegin: TYPE+ = posreal
TEnd: TYPE+ = posreal
TWindow: TYPE+ = posreal
TArrival: TYPE+ = posreal
TReady: TYPE+ = posreal
TDeadline: TYPE+ = posreal
TMaxComp: TYPE+ = posreal
task_type: TYPE+ = {primary, backup}
backup_task_status: TYPE = {active, passive, notbackup}
Processor_id: TYPE+ = posnat
Tfreetime: TYPE+ = posreal
task_property: TYPE+ = [# ArrTime:TArrival,
ReadyTime:TReady, BeginTime:TBegin, EndTime:TEnd,
Deadline: TDeadlineMaxCompTime:TMaxComp, TaskWindow:TWindow#]


task_vector: TYPE+ = [task_id → task_property]
task_assignment: TYPE+ = [#TaskId: task_id,
TaskType: task_type,

TaskStatus: backup_task_status #]
```

Next, we declare a *Processor* type. Every processor has an account of the free time available in it. *AllocateProcessor* defines a task assignment on a processor. Another key type definition is that of *Processor_database* which provides a set of tasks assigned on a processor. We also initialize the processor database.

```
ProcessorProperty: TYPE+ = [# freetime: Tfreetime #]
  Processor_Vector: TYPE+ = [Processor_id → ProcessorProperty]
```

```
AllocateProcessor: TYPE = [task_assignment → Processor_id]
processor_assignment: TYPE = [set[task_assignment]]
Processor_database: TYPE = [Processor_id → processor_assignment]
Initialise_database: processor_assignment = emptyset[task_assignment]
```

After having defined these basic types, we next formalize various constraints on task characteristics. The constraints are implemented by using *axioms*. The different axioms are:

- The begin time should always be less than or equal to the end time of the tasks*(Min_Max_AX)*
- The deadline of the task should always be less than the ready time of the task*(Deadline_ready)*
- The task window is defined as the time between the ready time of the task and the deadline of the task*(Win)*

```
AnyTask: task_vector
  TaskId1: VAR task_id
  TMin_Max_AX: AXIOM
    ∀ (TaskId1): BeginTime(AnyTask(TaskId1)) ≤ EndTime(AnyTask(TaskId1))
  TDeadline_ready: AXIOM
    ∀ (TaskId1): Deadline(AnyTask(TaskId1)) > ReadyTime(AnyTask(TaskId1))

  Twin: AXIOM
    ∀ (TaskId1):
      TaskWindow(AnyTask(TaskId1)) =
```

$$\text{Deadline(AnyTask(TaskId1))} - \text{ReadyTime(AnyTask(TaskId1))}$$

It should be noted that the task model formalization cannot be reused because the basic task set changes from periodic to aperiodic, however the new formalization contains only the new parameters and constraints for aperiodic tasks added in the exisiting task model specification.

4.2.2 Formalization of Primary Backup

The primary backup model followed by the dependable aperiodic task scheduling protocol is same as the model utilized by the dependable periodic tak scheduling protocols. We reuse the primary backup component which has already been specified earlier.

The primary backup component which was specified earlier is as follows:

```
PrimaryBackup: THEORY
 BEGIN

  IMPORTING rate_monotonic
  primary_task: TYPE = {n: task_assignment | TaskType(n) = primary}
  backup_task: TYPE = {n: task_assignment | TaskType(n) = backup}
  T: task_vector
  m: TaskInstance
  num: VAR task_assignment
  PR: VAR primary_task
  BK: VAR backup_task
  pro: AllocateProcessor
  db: Processor_database
  PBAX1: AXIOM
    ∀ (BK, (PR: primary_task | TaskId(PR) = TaskId(BK))):
      Execution (T (TaskId (BK) ) )  ≤  Execution (T (TaskId (PR) ) )
  PBAX2: AXIOM
    ∀ (BK, (PR: primary_task | TaskId (PR) = TaskId (BK) ) ) :
      Period (T (TaskId (PR) ) )  =  Period (T (TaskId (BK) ) )
  PBAX3: AXIOM
    ∀ (BK, (PR: primary_task | TaskId (PR) = TaskId (BK) ) ) :
      pro (PR)  ≠  pro (BK)
get_status (n: task_assignment) : backup_task_status =
      (IF TaskType (n) = backup ∧
  (Period(T(TaskId(n))) − CWork(TaskId(n))) ≥ Period (T (TaskId (n) ) )
          THEN passive
       ELSE active
       ENDIF)
  Pri (j: Processor) : set [task_assignment] =
      {n: task_assignment | (n ∈ db(j)) ∧ TaskType (n) = primary}
  Act (j: Processor) : set [task_assignment] =
      {n: task_assignment | (n ∈ db(j)) ∧ TaskStatus (n) = active}
  pas (j: Processor) : set [task_assignment] =
      {n: task_assignment | (n ∈ db(j)) ∧ TaskStatus (n) = passive}
  Active_Recover (j, f: Processor) : set [task_assignment] =
      {n: task_assignment | (n ∈ Act(j)) ∧ (n ∈ Pri(f))}
  Passive_Recover (j, f: Processor) : set [task_assignment] =
      {n: task_assignment | (n ∈ pas(j)) ∧ (n ∈ Pri(f))}
  Rec (j, f: Processor) : set [task_assignment] =
      (Active_Recover(j, f) ∪ Passive_Recover(j, f))
  NonFaultyGrp (j: Processor) : set [task_assignment] =
      {assign: task_assignment | (assign ∈ (Pri(j) ∪ Act(j)))}
  OneFaultyGrp (j, f: Processor) : set [task_assignment] =
      {assign1: task_assignment | (assign1 ∈ (Pri(j) ∪ Rec(j, f)))}
  PBTH: THEOREM
    ∀ ((num: task_assignment | TaskType (num) = primary)) :
      TaskType (num)  ≠  backup
END PrimaryBackup
```

Aperiodic tasks are scheduled based on the arrival and deadlines of the task. The tasks arrive dynamically so schemes like rate monotonic cannot be utilized. This eliminates the need of a scheduling component in the specification. Besides this, the allocation is also not formalized separately because the dependable aperiodic task scheduling protocol chosen by us in the case study has a specific allocation schema for itself. However, as future research direction we seek to evolve a generic model of the allocation schema for this protocol.

4.3 Putting Components Together to Formalize the Dependable Aperiodic Task Scheduling and Allocation Protocol

We utilize the task model component and the primary backup component to formalize the allocation and scheduling schema of the protocol. The present specification demonstrates the basic allocation functionality of the protocol.

4.3.1 Task Allocation

The protocol assumes thats the tasks arrive dynamically in the system and are scheduled non-preemtively as they arrive. The tasks are independent and have no precedence constraints. The protocol ascertains that the tasks which can be scheduled in a fault tolerant manner are only accepted to be scheduled. This is done by formalizing the notion of the boolean function *SparecompTime* and *PrimaryAdd.* These functions values are used to determine the feasibility of scheduling the task in a processor.

```
FTApSch: THEORY

 BEGIN

  IMPORTING PrimaryBackup

  f: Processor_id
  NoPr: posnat
  MinNoPr: posnat
  Pr_AX: AXIOM NoPr > MinNoPr
  SparecompTime?(AnyTask: task_vector, TaskId1: task_id, AnyProcessor: Pro-
cessor_Vector,

                       Processid1: Processor_id) :

  bool = (2 × freetime(AnyProcessor(Processid1)) ≥

            MaxCompTime(AnyTask(TaskId1)))
  PrimaryADD?(AnyTask: task_vector, TaskId1: task_id, AnyProcessor: Proces-
sor_Vector,

                    Processid1: Processor_id) :

  bool = SparecompTime?(AnyTask, TaskId1, AnyProcessor, Processid1)
```

The next step in the allocation is to assign a processor to the primary task. The primary task is allocated a processor based on the following constraints:

1. The task is guaranteed to complete if the processor fails at any instant of time.
2. The task is guaranteed to complete even if the backup task's processor fails , provided that the primary task's processor recovers before the failure.

The previously defined boolean functions are utilized to formalize the primary task addition procedure(*add_pri*).

```
add_pri(AnyTask: task_vector, TaskId1: task_id, AnyProcessor: Processor_Vector,
        Processid1: Processor_id, n: task_assignment,
((k: posnat | $k \geq$ MinNoPr $\wedge$ $k \leq$ NoPr)),
        db: Processor_database) :
  RECURSIVE Processor_database =
    IF PrimaryADD?(AnyTask, TaskId1, AnyProcessor, Processid1)
      THEN db WITH [(k) := $(\mathrm{db}(k) \cup \{n\})$]
    ELSE IF (NoPr $- k$ = 0)
           THEN db
         ELSE add_pri(AnyTask, TaskId1, AnyProcessor, Processid1,
  n, k+1, db)
         ENDIF
    ENDIF
     MEASURE NoPr $- k$
```

The backup task addition is influenced by the following constraints:

1. The primary and backup of a single task cannot reside in the same processor *(C4check?)*
2. The backup task should also be scheduled based on the *SparecompTime* and *PrimaryAdd* boolean function
3. If the primary task of two different tasks are scheduled in the same processor their backups should not be scheduled in a single processor.

The first constraint is satisfied by the primary backup component. The other two contraints are formalized below. It should be noted that the active backup scheduling assigns (*add_act*) two tasks to a processor. This is done to ascertain that the third constraint is satisfied.

```
C4check? (AnyTask: task_vector, $n_1$, $n_2$: task_assignment, $j$: Processor_id) :
 bool =

        ((($n_1 \in$ db($j$))) $\wedge$ TaskType ($n_1$) = primary) $\wedge$

         ((($n_2 \in$ db($j$))) $\wedge$ TaskType ($n_2$) = primary)
  ActiveBacAdd? (TaskId1, $k$: Processor_id, AnyProcessor: Processor_Vector) :
  bool =

        SparecompTime? (AnyTask, TaskId1, AnyProcessor, $k$)
  add_act ( ( ($n_1$) ) , ( ($n_2$: task_assignment | TaskStatus ($n_2$) = active) ) ,

  ( ($k$:posnat | $k \geq$MinNoPr$\wedge k \leq$ NoPr) ) , db: Processor_database,
   AnyTask: task_vector, $j$, TaskId1: task_id,
    AnyProcessor: Processor_Vector) :

  RECURSIVE Processor_database =

    IF SparecompTime? (AnyTask,
    TaskId1, AnyProcessor, $j$)

     $\wedge$ C4check? (AnyTask, $n_1$, $n_2$, $j$)

      THEN db WITH [ ($k$) := (db($k$) $\cup$ {$n_1$})]

    ELSE IF C4check? (AnyTask, $n_1$, $n_2$, $j$)

           THEN db WITH [ ($k$) := (db($k$) $\cup$ {$n_2$})]
         ELSE IF (NoPr $- k$ = 0)
               THEN db
              ELSE add_act ($n_1$, $n_2$, $k+1$, db, AnyTask, $j$,
      TaskId1, AnyProcessor)
              ENDIF
         ENDIF
    ENDIF
    MEASURE  NoPr − k
```

Finally, we specify the passive backup addition process (*add_pas*) which is done similarly as the active backup addition process.

```
PassiveBacAdd? (TaskId1, k:  Processor_id, AnyProcessor: Processor_Vector) : bool=
        SparecompTime? (AnyTask,  TaskId1,  AnyProcessor,  k)

   add_pas ( ( (n1) ) ,  ( (n2:  task_assignment  |  TaskStatus (n2)  =  passive) ) ,
     ( (k:  posnat  |  k  ≥  MinNoPr  ∧  k  ≤  NoPr) ) ,  db:  Processor_database,  Any-
Task:  task_vector,
                j,  TaskId1:  task_id,  AnyProcessor:  Processor_Vector) :
   RECURSIVE  Processor_database  =
      IF
      SparecompTime? (AnyTask, TaskId1, AnyProcessor, j)  ∧
      C4check? (AnyTask, n1, n2, j)
         THEN  db  WITH  [ (k)  :=  (db(k) ∪ {n1})]
      ELSE  IF  C4check? (AnyTask,  n1,  n2,  j)
                THEN  db  WITH  [ (k)  :=  (db(k) ∪ {n2})]
             ELSE  IF  (NoPr − k  =  0)
                      THEN  db
                    ELSE
add_pas (n1,  n2,  k + 1,  db,  AnyTask,  j,
 TaskId1,  AnyProcessor)
                    ENDIF
             ENDIF
      ENDIF
       MEASURE  NoPr − k
```

We next specify a task addition function *AddTaskN* which adds the primary tasks to the task database.

```
AddTaskN (AnyTask:  task_vector,  TaskId,  TaskId1:  task_id,
 AnyProcessor:  Processor_Vector,
       p:  Processor_id,  ( (n:  task_assignment  |  TaskType (n)  =  primary) ) ,
       ( (k:  posnat  |  k  ≥  MinNoPr  ∧  k  ≤  NoPr) ) ,
       db:  Processor_database) :
   Processor_database  =  add_pri (AnyTask, TaskId,  AnyProcessor,  p,  n,  k, db)
```

4.3.2 Verification of the Dependable Aperiodic Task Scheduling and Allocation Protocol

Using of the different theories to specify the dependable aperiodic task allocation and scheduling policies are verified by a theorem formulated in the FTApSch module. The following property is verified:

1) The backup task becomes active only when the primary task fails. The following theorem verifies that whenever a primary task is executing the backup does not start running.

CheckBack: THEOREM
$\forall$ (AnyTask: task_vector, TaskId: task_id, TaskId1: task_id,
AnyProcessor: Processor_Vector,
n: task_assignment, n_0: task_assignment, k: Processor_id,
((p: posnat | $k \geq$ MinNoPr $\wedge$ $k <$ NoPr $- 1$)),
((db: Processor_database
| db = AddTaskN(AnyTask, TaskId, TaskId1, AnyProcessor, k, n,
p, db))),
TaskId: task_id, AnyProcessor: Processor_Vector):
(TaskId(n) = b2n(($n \in$ db(k))) $\wedge$ TaskType(n) $\neq$ primary) $\supset$
(TaskStatus(n) $\neq$ passive)

4.3.2.1 Verification of Dependable Aperiodic Task Allocation and Scheduling Protocol: Results

The two theorems discussed earlier utilize the previously defined theories inorder to be proven. The detailed proving is shown in A The theorems were proven successfully and the proof summary in the mechanized proving using PVS is illustrated below:

```
Proof summary for theory FTApSch
add_pri_TCC1..................proved - complete (0.33s)
add_pri_TCC2..................proved - complete (1.14s)
add_pri_TCC3..................proved - complete (0.24s)
add_act_TCC1..................proved - complete (0.96s)
```

```
add_act_TCC2..................proved - complete (0.29s)
add_pas_TCC1..................proved - complete (0.91s)
add_pas_TCC2..................proved - complete (0.26s)
CheckBack.....................proved - complete (0.40s)

Theory totals: 8 formulas, 8 attempted, 8 succeeded (4.53 s)
Grand Totals: 9 proofs, 9 attempted, 9 succeeded (4.54 s)
```

We have demonstrated that the previously specified building blocks for FTRMFF protocol which is used to allocate and schedule periodic tasks, can be reused to specify another scheduling protocol for aperiodic tasks. The main objective was to show the reduced effort in formalization process and in turn propose a library of components that form the basic building blocks of scheduling protocols which can be reused to verify and specify all fault tolerant real time allocation and scheduling protocols. In in the next chapter we present the discussion that follow from this thesis.

Chapter 5

Discussions and Conclusions

Present day embedded systems are becoming increasingly dependent on large and complex real time scheduling protocols whose malfunctioning could have serious consequences. However, the size and complexity of modern day software systems make it almost impossible to avoid errors in the specification and implementation of dependable real time scheduling protocols. Consequently, tools and methods that can improve the specification and verification process are crucial in the protocol development. Traditional techniques and tools are not really able to cope with very complex multiprocessor system protocols. This is even more so the case for real-time systems, where timing aspects add further complexity to the system. Formal techniques which provide a rigorous and traceable framework, have been used to specify and verify such protocols. However, formal analysis typically requires intensive effort for both specifying and verifying each specific protocol. Most of these formal theories for real-time scheduling have been developed without much regard for their further reuse, and hence, much of formal specifications and their proof constructs in general are difficult to reuse to verify or analyze similar or related protocols. This, in our viewpoint, limits the wide acceptance of formal techniques in the design and development of dependable real-time systems.

5.1 Proposed Methodology

We have illustrated how various attributes and associated formal theories of different building blocks can be utilized in formulating a new protocol. We emphasize that the intent of our modular approach is to provide for reusability of fundamental building blocks across protocols. The key idea is that if a library of system/task/fault model and supporting building blocks can be formulated, then the overall formal models of fault tolerant real time protocols can get built with an easier and systematic reuse of functions.

We have formalized the FTRMFF protocol and show the viability to perform a rigorous verification of a moderately complex allocation and scheduling mechanism. Reusing the building blocks of the FTRMFF protocol, we specify and verify the FTRMNF protocol and also a dependable aperiodic task scheduling and allocation protocol. We show the rigourness involved in the verfication process by using a mechanized theorem proving mechanism provided by PVS. We also demonstrate the reduction in effort in the specification and verification process by reusing the building blocks.

Specifically, we introduce the notion of making a library of building blocks for all fault tolerant real time scheduling protocols. We also demonstrate the method of deriving the protocol specification from the building blocks and verifying the protocol in a mechanized manner using the PVS tool. This would imbibe a great deal of rigorness in the protocol verification with reduced effort and efficient delivery of the protocol services.

5.2 Benefits and Applications

The modular approach for formalization of real time fault tolerant scheduling protocols proposed in this thesis can be used in various areas involving protocol design and validation. We next highlight some key applications where this methodology could be used:

1. The set of library components can be used to specify a protocol as demonstrated by us. A rigorous development of the specifications for the library components would result in a descriptive specification of the resulting composite protocol. The actual implementation of the protocol can be obtained by using a tool which can generate code from the specification.

2. Any protocol resulting from the library complies to the specification prescribed for it. This protocol can be used as a standard to compare other implementations of the same protocol and a standard set of test cases can also be generated from it.

3. The other important benefit that has been emphasized throughout the thesis is reusability. Any effort done to formally specify and verify a protocol would reduce the effort to formally verify and specify any other related protocol subsequently designed and developed.

5.3 Limitations and Improvements

The present implementation serves as a demonstration of the theory proposed by us in this thesis. We have modeled the protocols in a very abstract manner so as to show the basic functionalities only. The present implementation does not address many operational and run time issues present in the actual protocol specification. We have specified only the basic working features to demonstrate our theory. This limits our study to only a few conformance tests to support the theory proposed by us. The theorems which are used to do mechanized verification of the specification presently verify the basic conditions only. However, the same results would follow for a detailed implementation of the protocols.

The present specification can be made more robust by descriptively specifying the component features and run time details. We also acknowledge that the effectiveness or generic applicability of the approach depends on an extensive suite of formal theories covering a wide spectrum of scheduling and heuristic allocation policies appearing in the literature.

5.4 Future Research Directions

The future research direction would focus into detailed component specification as well as introducing run time operational features in the specification. We also intend to propose a methodology for mapping the requirements to a set of rules which could be translated further into theorems in PVS. We also plan to use this model for diagnosis process, where the theorem prover could provide inputs to the diagnostic tool to trace out an error.

Specifically, our outlook is to develop a general theoretical framework where one can potentially plug-in formal theories of a specific scheduling policy, and subsequently perform analysis of algorithms which are based on it. We are also looking into parameterizing these formal theories so that the priority assignment policy, quality-of-service attributes, or the status (active/passive) of a subset of backup tasks can be passed as a parameter to the formal theory.

Appendix A

Appendix I

We present the mechanized proof steps for the TestDb1 theorem formalized in the thesis. For the purpose of brevity in the thesis we limit the detailed mechanized proving description for only this theorem.

A.1 Proof Steps of FTRMFF

A.1.1 Theorem TestDb1

```
pvs(9):
    TestDb1 :
    |——
    {1} FORALL (n: task_assignment,
    (k: Processor | k >= MinNoPr AND k < NoPr - 1),
    db: Processor_database | db = AddTaskN(n, k, db),
    TaskId: task_id):
    (TaskId(n) = b2n(member(n, db(k)))) IMPLIES
    (task_priority(TaskId) < task_priority(TaskId + 1))
```

```
Rerunning step: (lemma "RMA_Orderingrev")
Applying RMA_Orderingrev
this simplifi es to:
TestDb1 :
{-1} FORALL (TaskId1):
Number / Period(AnyTask(TaskId1)) <
Number / Period(AnyTask(TaskId1 + 1))
|-----
[1] FORALL (n: task_assignment,
(k: Processor | k >= MinNoPr AND k < NoPr - 1),
db: Processor_database | db = AddTaskN(n, k, db),
TaskId: task_id):
(TaskId(n) = b2n(member(n, db(k)))) IMPLIES
(task_priority(TaskId) < task_priority(TaskId + 1))
Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifi es to:
TestDb1 :
{-1} FORALL (TaskId1):
Number / Period(AnyTask(TaskId1)) <
Number / Period(AnyTask(1 + TaskId1))
|-----
{1} FORALL (n: task_assignment,
(k: Processor | k >= MinNoPr AND k < NoPr - 1),
db: Processor_database | db = AddTaskN(n, k, db),
TaskId: task_id):
(TaskId(n) = b2n(member(n, db(k)))) IMPLIES
(task_priority(TaskId) < task_priority(1 + TaskId))
Rerunning step: (grind)
```

```
member rewrites member(n, db(k))
to db(k)(n)
b2n rewrites b2n(db(k)(n))
to IF db(k)(n) THEN 1 ELSE 0 ENDIF
task_priority rewrites task_priority(TaskId)
to Number / Period(AnyTask(TaskId))
task_priority rewrites task_priority(1 + TaskId)
to Number / Period(AnyTask(1 + TaskId))
MinNoPr rewrites MinNoPr
to 1
NoPr rewrites NoPr
to 10
TotalWork_Pri_bac rewrites TotalWork_Pri_bac(time0, time1)
to W1(time0, time1) + W2(time0, time1)
FTRMFF_non_faulty? rewrites FTRMFF_non_faulty?(k!1)
to FORALL (number: task_assignment | member(number, NonFaultyGrp(k!1))):
W1(time0, time1) + W2(time0, time1) <= 1
NoFaultCTT? rewrites NoFaultCTT?(n!1, k!1)
to NonFaultyGrp(k!1) = union(singleton(n!1), NonFaultyGrp(k!1)) IMPLIES
(FORALL (number: task_assignment
| member(number, NonFaultyGrp(k!1))):
W1(time0, time1) + W2(time0, time1) <= 1)
FTRMFF_one_fault? rewrites FTRMFF_one_fault?(k!1, f)
to FORALL (n: task_assignment | member(n, OneFaultyGrp(k!1, f))):
W1(time0, time1) + W2(time0, time1) <= 1
OneFaultCTT? rewrites OneFaultCTT?(n!1, k!1, f)
to OneFaultyGrp(k!1, f) = union(singleton(n!1), OneFaultyGrp(k!1, f))
IMPLIES
(FORALL (n: task_assignment | member(n, OneFaultyGrp(k!1, f))):
```

```
W1(time0, time1) + W2(time0, time1) <= 1)
PrimaryADD? rewrites PrimaryADD?(n!1, k!1)
to (NonFaultyGrp(k!1) = union(singleton(n!1), NonFaultyGrp(k!1)) IMPLIES
(FORALL (number: task_assignment
| member(number, NonFaultyGrp(k!1))):
W1(time0, time1) + W2(time0, time1) <= 1))
AND
(OneFaultyGrp(k!1, f) = union(singleton(n!1), OneFaultyGrp(k!1, f))
IMPLIES
(FORALL (n: task_assignment | member(n, OneFaultyGrp(k!1, f))):
W1(time0, time1) + W2(time0, time1) <= 1))
FTRMFF_one_fault? rewrites FTRMFF_one_fault?(k!1, pro(n!1))
to FORALL (n: task_assignment | member(n, OneFaultyGrp(k!1, pro(n!1)))):
W1(time0, time1) + W2(time0, time1) <= 1
OneFaultCTT? rewrites OneFaultCTT?(n!1, k!1, pro(n!1))
to OneFaultyGrp(k!1, pro(n!1)) =
union(singleton(n!1), OneFaultyGrp(k!1, pro(n!1)))
IMPLIES
(FORALL (n: task_assignment
| member(n, OneFaultyGrp(k!1, pro(n!1)))):
W1(time0, time1) + W2(time0, time1) <= 1)
ActiveBacAdd? rewrites ActiveBacAdd?(n!1, k!1)
to (NonFaultyGrp(k!1) = union(singleton(n!1), NonFaultyGrp(k!1)) IMPLIES
(FORALL (number: task_assignment
| member(number, NonFaultyGrp(k!1))):
W1(time0, time1) + W2(time0, time1) <= 1))
AND
(OneFaultyGrp(k!1, pro(n!1)) =
union(singleton(n!1), OneFaultyGrp(k!1, pro(n!1)))
```

```
IMPLIES
(FORALL (n: task_assignment
| member(n, OneFaultyGrp(k!1, pro(n!1)))):
W1(time0, time1) + W2(time0, time1) <= 1))
PassiveBacAdd? rewrites PassiveBacAdd?(n!1, k!1)
to OneFaultyGrp(k!1, pro(n!1)) =
union(singleton(n!1), OneFaultyGrp(k!1, pro(n!1)))
IMPLIES
(FORALL (n: task_assignment
| member(n, OneFaultyGrp(k!1, pro(n!1)))):
W1(time0, time1) + W2(time0, time1) <= 1)
AddTaskN rewrites AddTaskN(n!1, k!1, db!1)
to IF TaskType(n!1) = primary THEN add_pri(n!1, k!1, db!1)
ELSE IF TaskStatus(n!1) = active THEN add_act(n!1, k!1, db!1)
ELSE add_pas(n!1, k!1, db!1)
ENDIF
ENDIF
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
Run time = 2.04 secs.
Real time = 3.44 secs.
```

Bibliography

[Alti2002] Altisen K., Goessler G., Sifakis J., Scheduler modeling based on the controller synthesis paradigm, Journal of RTS, No.23, pp.55-84, 2002.

[Alur1996] Alur R., Henzinger T.A., Ho P.H., Automatic symbolic verification of embedded systems, IEEE Trans. on SE, 22(3), pp.181-201, 1996.

[Ande1982] Anderson T., Lee P.A., Fault Tolerance Terminology Proposals, Proceedings of the 12th International Symposium on Fault-Tolerant Computing, pp. 29-33,1982

[Ande2001] Andersson B., Baruah S., Jansson J.,Static-priority scheduling on multiprocessors, In Proceedings of the IEEE Real-Time Systems Symposium , December 2001.

[Aviz2000] Avizienis A., Laprie J, Randell B., Fundamental Concepts of Dependability, 3rd Information Survivability Workshop, (ISW-2000) , Boston, Massachusetts, October 24-26, 2000.

[Baru2003] Baruah S., Goossens J., Rate-monotonic scheduling on uniform multiprocessors, Proceedings of the Twenty-Third International Conference on Distributed Computing Systems, pp. 360-366 (ICDCS03), Providence, Rhode Island. May 2003.

[Bert1999] Bertossi A.A., Mancini L.V., Rossini F., Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems, IEEE Transactions on Parallel and Distributed Systems, vol. 10, n. 9, pp. 934-945,1999.

[Brab1999] V. Braberman V., Felder M., Verification of real-time designs, LNCS 1687, pp.494-510, 1999.

[Dert1974] Dertouzos M., Control robotics: The procedural control of physical process, in proceedings of IFIP congress, 1974.

[Dert1989] Dertouzos M.L., Mok A., Multiprocessor Online Scheduling of Hard-Real-Time Tasks, IEEE Transactions on Software Engineering, v.15 n.12, p.1497-1506, December 1989

[Dhall1978] Dhall S.K., Liu C.L., On a Real-Time Scheduling Problem, Operations Research, vol. 26, no. 1, pp. 127–140, Jan.1978.

[Dute2000] Dutertre B., Formal Analysis of the Priority Ceiling Protocol, Proc. Of RTSS-21, 2000.

[Funk2002] Funk S., Goossens J., Modifying EDF Uniform Multiprocessors, Proceedings of the 14th Euromicro Conference on Real- Time Systems, 2002.

[Ghos1997a] Ghosh S., Melhem R., Mossé D., Fault-Tolerant Rate-Monotonic Scheduling, In Dependable Computing for Critical Applications (DCCA-6), IEEE Computer Society, 1997.

[Ghos1997b] Ghosh S., Melhem R., Mosse D., Fault-Tolerance through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems, IEEE Trans. on Parallel and Distributed Systems, vol 8, no 3, pp. 272-284, 1997.

[Ghos1998] Ghosh, S., R. Melhem, D. Mossé, J. Sarma, Fault-Tolerant Rate-Monotonic Scheduling, Real-Time Systems, Vol. 15, Kluwer Academic Publisher, pp. 149-181, 1998.

[Kettl1995] Kettler K. A., Lehoczky J. P., Strosnider J. K., Modeling bus scheduling policies for real-time systems, Real-Time Systems Symposium, p 242-253 , 1995.

[Klei1993] Klein M.H., et al. A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems, Kluwer Academic Publishers, Boston, 1993.

[Krish1997] Krishna C.M., Shin K.G., Real-Time Systems, McGraw Hill, 1997.

[Lee1990] Lee P.A., Andersson T. (ed.), Fault Tolerance - Principles and Practice, Dependable Computing and Fault-Tolerant Systems Series, Vol. 3, Second edition, Springer Verlag, 1990

[Leho1989] Lehoczky J.,Sha L., Ding Y., The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. Proc. of RTSS, pp. 166–171, 1989.

[Liu1973] Liu C.L., Layland J.W., Scheduling Algorithms for Multiprogramming in a Hard-Real-Time, Environment Journal of the Association for Computing Machinery, Vol. 20, No. 1, January 1973

[Liu1999] Liu Z.,Joseph M., Specification and Verification of Fault-Tolerance, Timing and Scheduling, TOPLAS, 21(1), pp. 46–89, 1999.

[Liu2000] Liu J.W.S, Real-Time Systems, Prentice Hall, 2000.

[Mok1983] MOK A., Fundamental design problem of distributed systems for the hard real-time environment, Dissertation M.I.T,Cambridge, May 1983.

[Owre1995] Owre S., Rushby J., Shankar N., F. von Henke, Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS, IEEE Transactions on Software Engineering, Vol. 21, No 2, pp. 107-125, 1995.

[Rand1978] Randell B., Lee P.A., Treleaven P.C., Reliability Issues in Computing System Design, Computing Surveys, Vol. 10, pp. 123-165, 1978.

[Stan1988] Stankovic J., Misconceptions about real-time computing: A serious problem for next generation systems. IEEE Computer 21, 10, pp. 10-19, 1988.

[Sinh1999] Sinha P., Suri N., On the Use of Formal Techniques for Analyzing Dependable Real-Time Protocols, In Proceedings of the 20 th IEEE Real-Time Systems Symposium (RTSS-99), Phoenix, AZ,1999.

[Sinh2001] Sinha P.,Suri N., Modular Composition of Redundancy Management Protocols in Distributed Systems: An Outlook on Simplifying Protocol Level Formal Specification & Verification, Proceedings of ICDCS-21, pp 255-263, 2001.

[Vest2000] Vestal S., Modeling and verification of real-time software using extended linear hybrid automata, 5th NASA Workshop, pp.95-106, 2000.

[Yama2002] Yamane S., Refinement Theory of Embedded systems based on Hybrid models. The 2002 IKE, pp.455-461, CSREA Press, 2002.

[Yama2003] Yamane, S., Deductive schedulability verification methodology of real-time software using both refinement verification and hybrid automata, Computer Software and Applications Conference, COMPSAC 2003. Pages:527 - 533, 2003.

[Yuhu1994] Yuhua Z., Chaochen Z.A, Formal Proof of the Deadline Driven Scheduler, Proc. of FTRTFT, LNCS–863, pp. 756–775, 1994.

[Zhan2001] Zhan, N., An intuitive formal proof for deadline driven scheduler, Journal of Computer Science and Technology, v 16, n 2, March 2001.