VERIFICATION OF FLEXRAY MEMBERSHIP PROTOCOL USING UPPAAL

by

VINODKUMAR SEKAR MUDALIAR

B.E., University of Pune, India, 2004

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2008

Approved by:

Major Professor
Dr. Mitchell Neilsen

Abstract

Safety-critical systems embedded in avionics and automotive systems are becoming increasing complex. Components with different requirements typically share a common distributed platform for communication. To accommodate varied requirements, many of these distributed real-time systems use FlexRay communication network. FlexRay supports both time-triggered and event-triggered communications. In such systems, it is vital to establish a consistent view of all the associated processes to handle fault-tolerance. This task can be accomplished through the use of a Process Group Membership Protocol. This protocol must provide a high level of assurance that it operates correctly. In this thesis, we provide for the verification of one such protocol using Model Checking. Through this verification, we found that the protocol may remove nodes from the group of operational nodes in the communicating network at a fast rate. This may lead to exhaustion of the system resources by the protocol, hampering system performance. We determine allowable rates of failure that do not hamper system performance.

Table of Contents

List of Figures

List of Tables

Acknowledgements

This work is a product of dedication, hard-work and devotion of mine, but these alone would have fallen short if not for the support, guidance, inspiration and advice of many others.

I am immeasurably indebted to my advisor, Dr. Mitchell L. Neilsen, for being a constant source of knowledge, insight, encouragement and support over the past two years. His intellectual contribution to this work, both directly and indirectly, has been invaluable to my progress with it. I wish to thank him for allowing me the discretion of walking into his office anytime that I had a question, and for giving me the opportunity to pursue my research under his guidance. He kept me going in my work in more ways than I can mention. His contribution to the clarity, notations and conciseness of this document is immense and I wish to thank him for the time and the effort that he has invested.

I wish to thank Dr. Gurdip Singh and Dr. Daniel Andresen for taking time out of their busy schedule and serving on my thesis committee. Their individual contribution towards my academic development has been immense and their kindness, expertise and lively enthusiasm have made studying enjoyable.

I would also like to thank my friends who have been a constant source of encouragement and support. And last, but definitely not the least, I would like to thank my family without whom I would have fallen astoundingly short of pursuing something so ambitious.

CHAPTER 1 - Introduction

In recent years, vehicles have been loaded with electronics which have increasingly defined the driving experience. Beginning with engine management and car audio, electronics have now penetrated all major systems in the vehicle, ranging from power train, body and chassis to driver-assistance systems and active and passive safety systems. A major concern with all of these electronics is safety. Vehicles are more prone to digress from their normal operation, in the event of sudden change, in how they operate. This is the reason why safety critical applications in automotive domains demand high reliability. In most applications, reliability is addressed by using fault-tolerant protocols. An example would be a process group membership protocol for FlexRay [1], which takes advantage of FlexRay's dual scheduling ability.

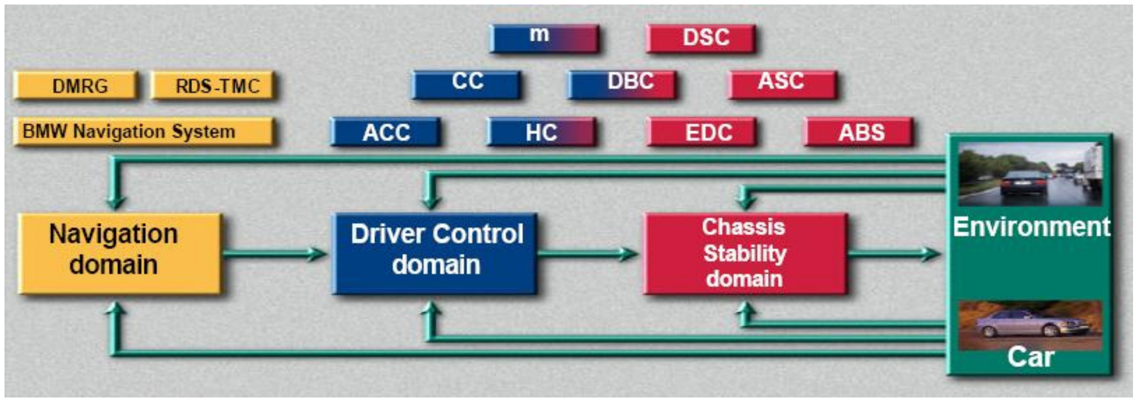
It may be observed that fault-tolerant distributed protocols are very subtle and informal arguments of correctness are prone to error. Such arguments are clearly insufficient for safety-critical applications which require a high level of assurance in their operation. Ideally, mathematical proofs, either manual or automatic, for correctness of protocol should be provided. An alternative formal method is model checking of the protocol. In this thesis we envisage to formally prove the correctness of a membership protocol for FlexRay using model checking.

1.1 Why FlexRay?

The first step towards use of electronics in the car was by introducing digitally controlled combustion engines with fuel injection and digitally controlled anti-lock brake systems (ABS). A lot of functionalities such as traction control (TCS), electronic stability control (ESP), and brake assistant (BA) helped having close synergy between mechanics and electronics. This yielded several benefits which included better fuel economy, and better vehicle performance in normal and adverse conditions.

Electronic and computer-based features continue to proliferate automotive industry, leading to widespread use of embedded real-time control networks. This is evident from Figure 1.1, which gives a snapshot of present electronics systems and architectures in cars.

Figure 1.1 [12] Snapshot of in-car E/E Systems



To gain additional performance means giving up mechanical linkages between driver and vehicle. The car industry has started using X-by-wire control. X-by-wire control will help make the car lighter.

Critical to X-by-wire designs are the safety components of software, hardware, and in particular, computer networks that coordinate vehicle functions. This new technology requires more bandwidth for the communication [2]. The average bandwidth needed for engine and chassis control is estimated to reach 1500 kb/s in 2008 as opposed to 765 kb/s in 2004 and 122 kb/s in 1994. The CAN protocol commonly used in the automotive industry, may soon be replaced by FlexRay, another protocol which provides the system advantages of higher bandwidth and support for both event-triggered (as CAN) and time-triggered communication.

1.2 Safety Critical Systems

A system is safety-critical if a failure of the system leads to consequences that are determined to be unacceptable. Thus, a system on which we depend for our well being is safety-critical. For example if the brake for one wheel of a car fails, it may cause the car to go out of control. Thus, the braking system is definitely a safety critical system.

Safety critical systems are called safety-related systems in the literature in the field. Neil Storey defines Safety related systems in his book ‘Safety Critical Computer System’:

A safety related system is one by which the safety of equipment or plant is assured.

Though a system may claim to be safe, it is necessary to prove that the system is safe. To believe that a system is going to be used in the real world, we need a proof of a safe and correct system behavior. Systems cannot be absolutely safe, but the safety level of a system must be sufficiently high for a specific task it's supposed to perform.

In the car example, when the brake for one wheel of a car fails the other brakes needs to be aware of this as soon as possible. They may then be able to compensate for the loss of one brake and stop the car almost as safely as with all brakes fully functioning.

We can foretell the systems behavior when it is predictable. Hence, predictability is one of the most desired properties for safety related systems.

1.3 Thesis Goals

The project creates a UPPAAL model that closely matches a real-life FlexRay communication in vehicles. We then extend the model to support fault-tolerance in the FlexRay protocol. The model will then be used to verify the protocol using model checking. It will be further used for verifying of industrial systems – in this case, embedded controllers with timing aspects.

CHAPTER 2 - Background

2.1 Fault Tolerance

The objective of the use of fault tolerance is to design a system in such a way that faults do not result in system failure. It is not possible to prepare for all contingencies; hence designing a system which is 100% safe is impossible. The aim of a fault tolerant system is to provide maximal sustained system dependability to make the system “safe enough”.

Today, the vehicles encompass more and more devices on board, which increase the chances of failure in the distributed system. The failures can grow arbitrarily large as number of components increase, hence we require the system to enter a state of graceful degradation. In this state the system will continues function in a decreased capacity when one or more components fail.

Use of fault tolerance varies between one application and another. It improves reliability, dependability and safety.

2.1.1Redundancy

Fault tolerance is achieved by using some form of redundancy. Redundancy may be implemented on many levels such as software redundancy, hardware redundancy, time (temporal) redundancy or information redundancy [2].

2.1.1.1 Information Redundancy

In a system containing information redundancy supplementary information in addition to that needed by the system in absence of faults is used. Examples of information redundancy include parity bits or CRC checksums.

2.1.1.2 Software Redundancy

Using software in addition to that required to implement a system in the absence of faults. Examples of these are membership algorithms.

2.1.1.3 Hardware Redundancy

Using hardware in addition to what is required to implement a system in the absent of faults. Examples of this are airplane onboard computers.

2.1.1.4 Time Redundancy

Using time in addition to what is required to implement a system in the absence of faults. Example of this are recalculating a result multiple number of times and then using majority voting to obtain the final result.

Many early fault tolerant systems duplicate hardware in such a way that if one module failed the rest of the system would still function normally. One example is a TMR (Triple Modular Redundancy) system where there are identical modules which produce the same output. A voting procedure was used to detect faulty units.

2.1.2 Reliability

When talking about safety related systems different terms such as reliability and availability are examples of topics. A safety critical system such as a brake-by-wire system in a car must be reliable. Storey defines reliability as “The probability of a component, or system, functioning correctly over a given period of time under a given set of operating conditions” [3]. One common definition is to denote reliability over time by R (t) and define it for a given number N of identical components that start operating at the same time as:

$$R\ (t) = n\ (t)/N$$

where n (t) is the number of components still operating correctly at time t. For a system consisting of different sets of redundant components one may use reliability models such as reliability diagrams to find the reliability of the entire system. The configuration of the system components, their individual reliability and the number of redundant units will all greatly influence the reliability, cost and performance of the system. When designing a dependable system this adds considerable complexity to finding the optimum configuration.

2.2 System Model

We consider a distributed system framework, comprised of a completely connected network of (N) nodes that supports both time-triggered and event- triggered communication. We consider the system communication model as:

A1. A path exists from each node to all other nodes (i.e., a completely connected network). A node issues a single message which is broadcast through its associated communication link.

A2. A non-faulty node can identify the sender of an incoming message, and can detect the discrepancy in an expected message. A faulty link can abrupt the message delivery but cannot correct an erroneous message.

A3. All processors execute different amounts of workload and determine the output value Vi through the use of a voting function.

A4. Node and link faults are considered indistinguishable: Links are considered as simple memory less interfaces between nodes.

A processor consists of one incoming link and one outgoing link. The incoming and outgoing links, broadcast message to receivers and receive message from other processors respectively. The protocol is non authenticated message passing; where the communication is organized into periodically repeated cycles of equal length. They contain a static and dynamic segment and network idle time. In addition, every message is checked for errors; thus, it is difficult for a malicious node to forge as a different processor. We assume that the processor supports clock synchronization.

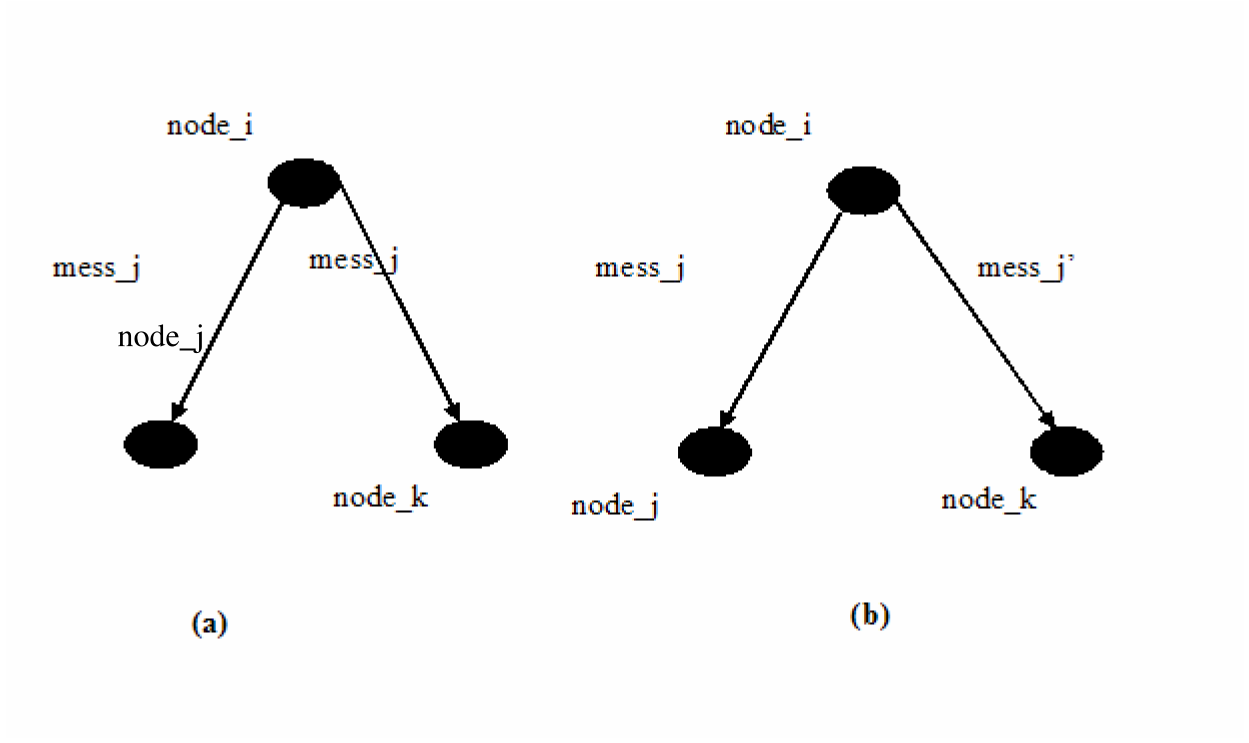
2.3 Fault Model

System failures are assumed to be due to inconsistent decisions or computations across the system. Thus, it is necessary to identify faulty nodes and prevent system failure in the presence of a specific number of faults. Fault models need to cover a range of fault types. Thus, flexibility in fault model helps in incorporating realistic system environment.

In [1], distinguishes failure into asymmetric and symmetric. Symmetric failures are perceived by all non-faulty nodes which occur due content failure in messages, process crashes in the system. For example a node receives a message which fails to match its checksum, then that node is considered to be faulty and removed from the membership group. An asymmetric failure causes the node to obtain an inconsistent view of the membership state. This occurs when a subset of nodes in the system receive message incorrectly.

The Hybrid Fault-Effects Model presented in [6, 7] is based on fault-detection mechanisms built in at the node level and on nodes exchanging their local opinion with other nodes in the system. Classification in HFM begins with the examination of fault states created by the dispersal of information. As shown in fig 2.1 general cases are possible when a node transmits information in the network. (a) all receivers obtain the same information i.e., mess_j (symmetric dispersal), or (b) receivers obtain different information i.e., mess_j and mess_j' (asymmetric dispersal).

Figure 2.1 Hybrid Fault Space (a) symmetric communication (b) asymmetric communication



The second aspect of the HFM is based on propagation effects of a faulty source. If an individual node is sufficient to detect the error, it is classified as a benign fault since the fault-effect is locally detectable. On the other hand, there are situations that require multiple nodes to exchange their syndrome information with each other in order to provide accurate diagnosis, i.e., globally detectable. For these cases, the values in a message appear to be plausible locally at a node; however, they can only be verified with a multiple exchange of information. After the exchange is completed, the plausible value may be determined to be value-faulty.

We need to ensure that distributed nodes or processes arrive at the same decisions and computational results in the presence of arbitrary faults. Thus, Byzantine [8] agreement algorithms are implemented.

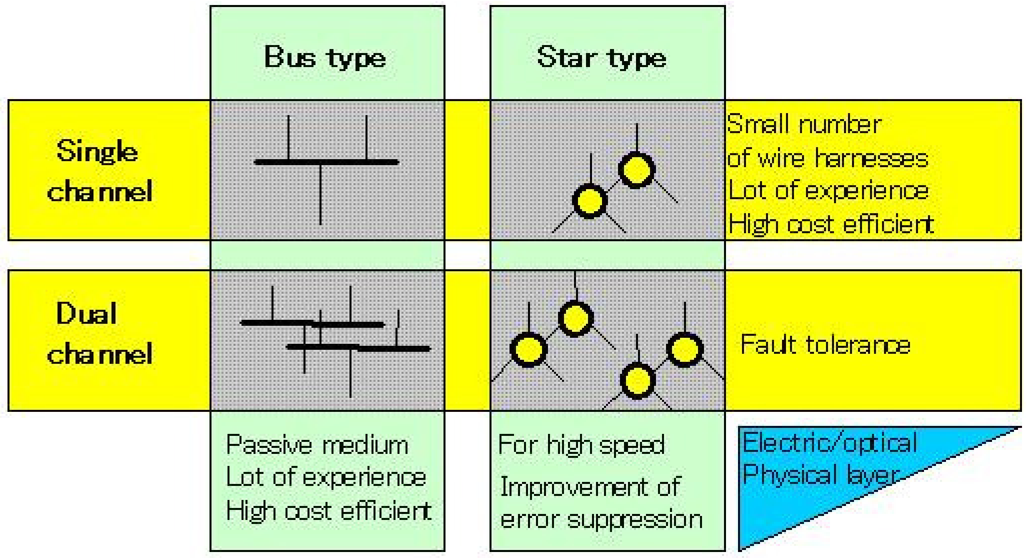
2.4 FlexRay

We need a communication system that handles applications limited by the bit rate of CAN. A system that offers redundancy in developing solutions which sends same messages several times or by providing two separate channels transmitting the same data. The system should be able to serve all future electronic functions in motor vehicles. In order to attain these objectives, a FlexRay consortium with it original members BMW, Daimler-Chrysler, Philips and Motorola was formed. The cooperation between some car manufacturers and electronics manufacturers resulted into a communication system called FlexRay. It is a system for advanced automotive control applications. The Consortium has grown to include some of the automotive industry's largest and most influential players, including Bosch, General Motors, and VW among others [4].

2.4.1 Overview

FlexRay is a network protocol based on TDMA scheme. The access to the medium is predetermined with time slots which structurally eliminates any possibility of message collision. It supports both time-triggered and event-triggered communication. The maximum bandwidth in FlexRay network is 10 Mbit/s. The two basic topologies used in FlexRay are the bus topology and the star topology. These topologies can be combined to form hybrid topologies over one or two channels. The support for dual channels improves fault tolerance in FlexRay protocol. Safety-Critical systems can use a second channel in order to transmit a redundant copy of the data on the first channel. It may setup to use either only one channel or two channels simultaneously when sending data. The minimum number of nodes in a FlexRay network is 2, and the maximum number of connected nodes is 64.

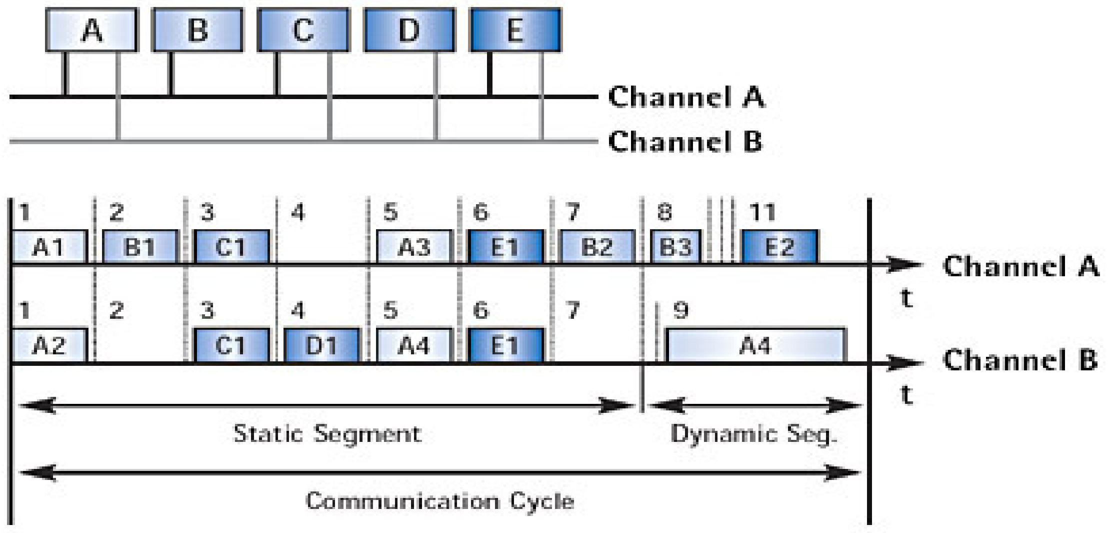
Figure 2.2 [5] Topologies used in FlexRay



Communication cycle

Communication in FlexRay takes place through repetitive communication cycles, each cycle consists of a static segment, a dynamic segment, an optional symbol window and finally a phase in which the network is in idle mode, called network idle time (NIT)

Figure 2.3[9] Different Segments making up a communication cycle



2.4.1.1 Static Segment

The part of the communication cycle in which access to the medium is monitored and controlled by a static time division multiple access principle is called the static segment in the FlexRay protocol. The static segment consists of a configurable number of slots, each slot containing room for the same amount of data. Messages are broadcasted to every other node in the network. When there is a need for communication which cannot be completed in one slot, the node is assigned several slots which give it a larger static piece of communication. It is also possible for a node to only listen to a channel by not being assigned to any slot at all. The slots that are not assigned to any node and the slots assigned to broken nodes will be silent. The static segment provides deterministic communication which defines the semantics of status messages and manages distributed systems.

2.4.1.2 Dynamic Segment

The dynamic segment is used to send event-triggered messages. This part of the communication cycle consists of mini slots which controls and monitors the access to the medium. This is also known as flexible time division multiple access (FTDMA). The dynamic segment is divided into minislots such that each slot has a unique frame id. Each mini-slot will be extended to a full-size dynamic slot when the node assigned to the current mini-slot has a message to transmit.

A minislot counter is used to provide collision avoidance in the dynamic segment. During a cycle the minislot counter value increases with a short time interval between ticks. A dynamic message is then sent when its frame id is equal to the current value of the minislot counter. When there is nothing to be sent in the dynamic segment of a cycle, then the minislots have same length. When a message is sent in the dynamic segment the duration of the minislot expands until the end of the message transmission. For example, in figure 2.2 the message F2 is being sent on channel two during most of the dynamic segment of that cycle. The message E2 can not be transmitted until the transmission of F2 has completed. In this way collision voidance is implemented in the dynamic segment. Two dynamic messages can never attempt to transmit at the same time on the same channel. This is the case because each message has a unique Frame ID. During the transmission of a message the value of the minislot counter is unchanged.

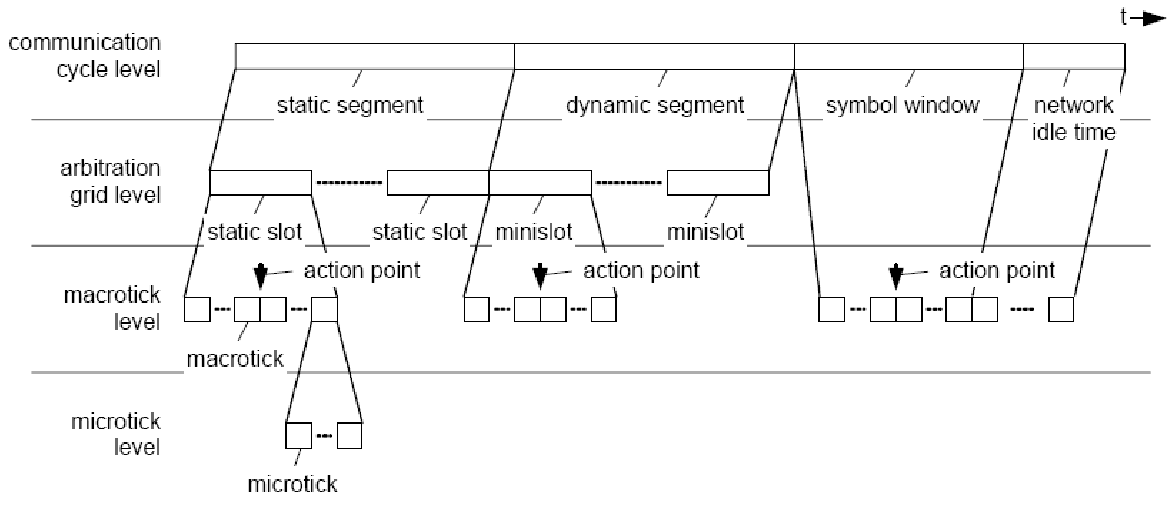
2.4.1.3 Network Idle Time

It is time at the end of each cycle. This phase is used for performing implementation specific related tasks NIT is used synchronization of timing in the protocol.

2.4.2 Timing Hierarchy

Time in a FlexRay node is represented by cycles, macroticks and microticks. The highest level, the communication cycle level, defines the communication cycle. A cycle builds on an integer number of macroticks. A macrotick is composed of an integer number of microticks. A microtick is a multiple of the clock tick of an oscillator in the communication controller. Since the tick length of oscillators in different communication controllers vary slightly, this has the signification that the length of a microtick is controller specific i.e. it may vary between different communication controllers.

Figure 2.4 [10] Time Hierarchy in FlexRay Controller



The FlexRay controller measures time on a local as well as global time basis. The local time in a node is defined as the time determined by using the node’s internal clock. The global time of a cluster is defined as the commonly agreed upon view of the passage of time inside a cluster. A global time is periodically determined through the exchange of messages, and the local time in each node is subsequently adjusted to match the global time.

2.4.3 Clock Synchronization

It is essential that the local time in the various nodes in a FlexRay network remains synchronized. Therefore the difference between local time in a node and the global time must not differ by any large amount. Any node straying too much from the local time too long is disconnected from the network. The time synchronization is done by the sending of a reference message in the static segment. The message is sent by the pre-selected synchronization nodes (nodes that have the sync bit in the header set).

2.5 UPPAAL

UPPAAL [11] is an open source tool for simulation and verification of real-time and embedded systems modeled as real-time components. It is jointly developed by the Uppsala University and Aalborg University.

It has a Java interface with verification engine written in C++. This tool helps verify systems that can be modeled as networks of timed automata extended with integer variables, structured data types, and channel synchronization. It also supports graphical modeling of internal component behavior as Uppaal timed automata, and hierarchical composition of components. We can also use the simulator to explore the dynamical behavior of components, and can be used to call the model-checker of Uppaal PORT to verify properties expressed in a subset of timed CTL.

UPPAAL has undergone several changes since it conception in 1995, most versions consisting of additions to their earlier counterparts. Some of the additions include, experiments and improvements include data structures, partial order reduction, symmetry reduction, a distributed version of Uppaal, guided and minimal cost reachability, work on UML Statecharts, acceleration techniques and new data structures and memory reductions.

Uppaal consists of three main parts: a description language, a simulator and a model-checker. The description language is a non-deterministic command language which serves as a modeling or design language to describe system behavior as networks of automata. The simulator is a validation tool which enables examination of possible dynamic executions of a system during early design stages. It provides an inexpensive means for fault detection prior to verification by the model-checker. The model-checker is used to check invariant and reachability properties by exploring the state-space of a system.

The model-checking in UPPAL is based on the theory of Timed Automata and its modeling language. The query language of UPPAAL which is a subset of CTL (computation tree logic) [3, 35] is used to specify the properties to be checked.

2.5.1 Timed Automata

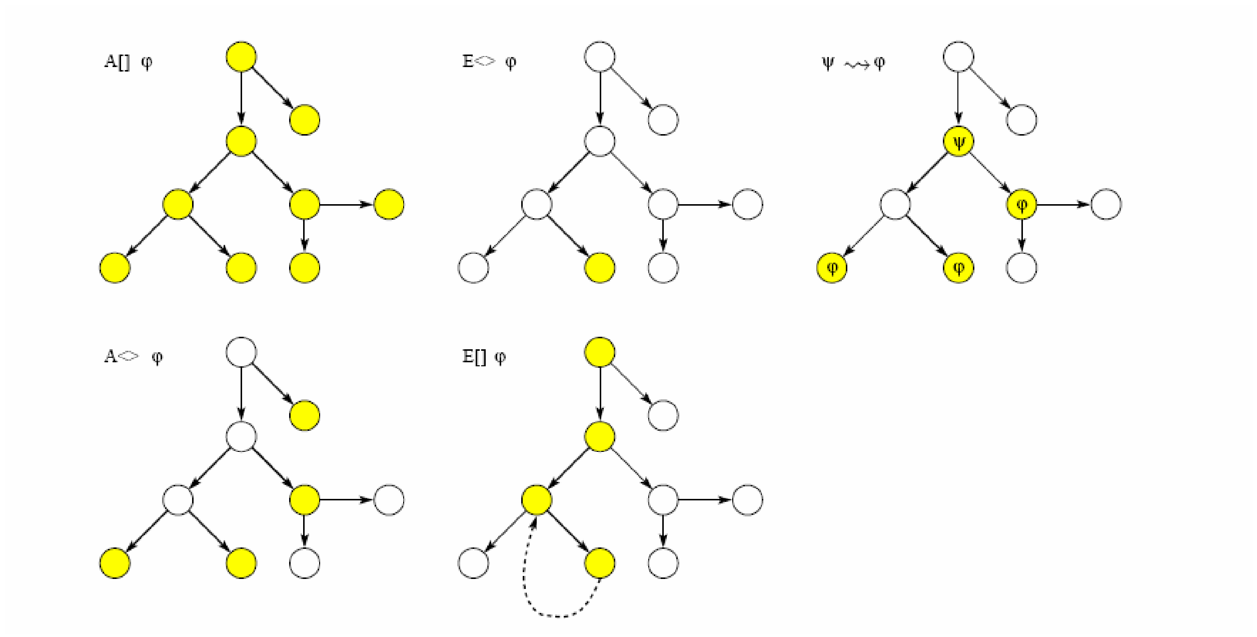
A timed automaton is a finite-state machine with clock variables. A system is modeled as a network of several such timed automata in parallel, with bounded discrete variables that are part of the state. These discrete variables can read, write and are subject to common arithmetic operation as in any programming language. A state of the system is defined by the locations of all automata, the clock constraints, and the values of the discrete variables.

2.5.2 UPPAAL Query Language

The model checker is used to verify the automata with respect to the given requirement specifications. The requirement specification is expressed in a formally well-defined and machine readable language. UPPAAL uses a simplified version of CTL for requirement specification for the model checker. State formulae are described as individual states, whereas path formulae quantify over paths or traces of the model. Path formulae can be further classified into reachability, safety and liveness.

We will describe each of the different path formulae supported by Uppaal.

Figure 2.5 [11] Path Formulae Supported in UPPAAL



2.5.2.1 State Formulae

A state formula is an expression which is evaluated for a state without looking at the behavior of the model. For example consider the expression, a = 10. This expression could be applicable for a state X of the automaton, that is true in a state whenever a equals 10. The syntax of state formulae is a superset of that of guards.

2.5.2.2 Reachability Properties

A reachable property is a property that can eventually hold true. A reachability property checks if a given state formula, possibly can be satisfied by any reachable state. They are often used in designing a model to perform sanity checks. A reachability property will not guarantee the correctness of the protocol but only validate the basic behavior of the model.

A special case would be the state formulae for a deadlock. The formula simply consists of the keyword deadlock and is satisfied for all deadlock states.

2.5.2.3 Safety Properties

Safety properties are properties which are required to hold at all times i.e. they are invariants. An example of a safety property would be the operating temperature of a nuclear plant which is always (invariantly) required to be under a certain threshold. In Uppaal these properties are formulated positively, e.g., something good is invariantly true.

2.5.2.4 Liveness Properties

Liveness property is a property which will hold true within a certain time bound. An example of a liveness property would be any message that has been sent should eventually be received, in any model of a communication protocol.

CHAPTER 3 - Membership Protocol

This section gives a detailed description of the membership protocol that has been modeled in this thesis. The membership protocol under consideration for distributed real-time systems supports both time-triggered and event triggered communication. The protocol consists of the algorithms that are described below.

3.1 Overview:

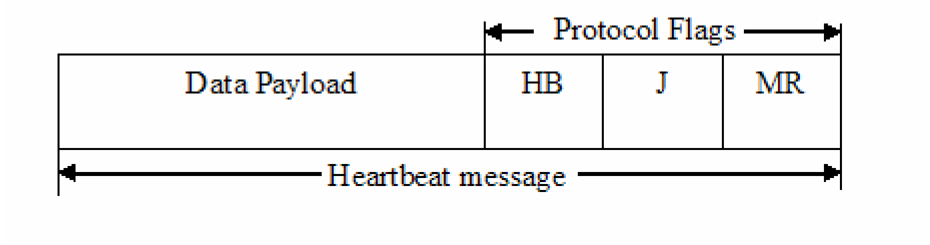
The membership protocol under consideration was proposed by Carl Bergenhem and Johan Karlsson in their article “A process Group membership Service for Active Safety Systems using TT/ET Communication Scheduling". In this thesis, this protocol has been modeled in Uppaal and verified.

A membership protocol is a membership service based on the idea that each node in the network creates its own view of the status of all other nodes in the network, i.e. which nodes are operational and which are not. A common view is formed based on the individual views of each of the nodes.

A common view is obtained by conducting a majority voting (poling), with a variety of random messaging by any/all nodes in the communication network. FlexRay being a time triggered protocol, each node in the network decides onto the absence/presence of a particular message on the network. This allows the protocol to decide whether a node is operational or not in the network. Thus when applied to a distributed real-time system, the protocol allows a group of operating real time processes to establish a "consistent" view of the operational status of the network. The protocol is developed to be applied to any application domain, but has been developed taking into consideration automotive active safety systems. A membership protocol would provide invaluable support for implementing fault tolerance and graceful degradation.

3.2 HeartBeat Message

Figure 3.1 Heartbeat Message



An important function of the membership protocol is the failure detection. The membership protocol should be able to detect the failure of a process which does not respond as expected. The FlexRay protocol, being a time triggered protocol simplifies this process, each node on the network decides whether a node is operational in the network. Each process is associated with a specific static slot in which it sends a message via the run time environment, a message once during each cycle. This message is also known as the heartbeat message. Each message in these static slots has a specification of timing for each node. Any deviation from the specified timing implies failure of the associated process, which is noticed in the run time environments of the nodes.

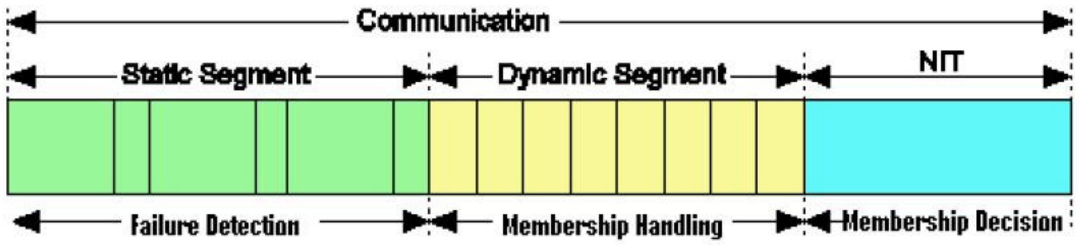
Conversely, each slot in has a fixed association with a process / node. A failure detection scheme is based on the presence of the "heartbeat" message as well as a contained status flag. The presence of the heartbeat message with the status flag on, implies that the associated process is correct and indirectly the run - time environment of the hosting node. It may also be noted that a node can be associated with several processes. Thus there can be several heartbeat messages associated with a node during each cycle. It may also be noted that a heartbeat message can be lost or corrupted during transmission. This only leads to faulty conclusion, indicating that the associated process is faulty.

The heartbeat message may also contain a data payload from the process, apart from the failure detection associated with it. This data payload may be consumed by the receiving process to perform its task. Each heartbeat message has 3 status flags associated with it, which are used for failure detection and the data payload carried by it from any communicating process to a receiving process. Each flag can be set or reset by the run - time environment .The flags are: heartbeat status (HB flag), join - request flag (J - flag), membership status flag (MR flag), see Fig 3.1.The HB flag is associated with the process of fault detection of the process in the communication network. The HB flag encodes one of the following states, concerning the corresponding process: dead, beating, signaled failure report. The status of the HB - flag when set to "beating", indicates that the sender membership service considers corresponding process to be a member. The HB flag when set to dead signifies that the associated process is not a member of the communication network. When the HB flag is set to signal failed report, it indicates that the payload of the message contains a failure report instead of the expected data from the sending process. A failure report is generated by the run - time environment, which may contain detailed information about the failure of one or many process.

3.3 Protocol

The protocol operation is divided into three phases that take place during different parts of a FlexRay cycle. The protocol proposed by [1] support nodes with multiple processes.

Figure 3.2 Membership Protocol in FlexRay Cycle



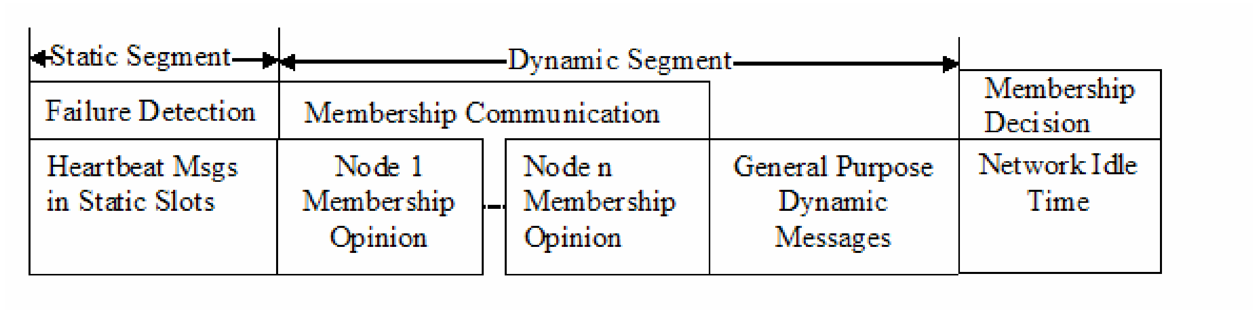
The basic function of the protocol is to form an opinion about the presence and the status of heartbeat messages. Each active node in the communication network broadcasts its opinion at the end of each communication cycle. Thus each active node in the network has complete information about the opinion of all the other nodes in the network. Thus each node can make a decision about the status of processes during the previous cycle. The local opinion of an active node may differ due to asymmetric failures .It may be noted that local opinions are inconsistent with the majority opinion .So asymmetric failures are manifested as content failures from the affected nodes. The most important function of the protocol is to differentiate between an asymmetric failure and failure at the receiver or the sender in the network.

The protocol is executed by each node in the communication network. When processing is completed by the protocol, the membership view is available at each node. It may be noted that the protocol messages must be send in each cycle, where each message may or may not be in a static slot, thus facilitating the absence of a event triggered communication scheme. Also, local opinions need to be exchanged between the nodes, in case of change in the system. In the event, that no local opinion needs to be exchanged, more network capacity is available for other event triggered messages to be sent on the network. The protocol tolerates failures as long as at-least a majority of nodes are active during two cycles.

3.3.1Basic Protocol

The operation of the protocol is divided into 3 phases: failure detection, membership communication, membership decision. Figure 3.3 shows the phases of the protocol.

Figure 3.3 Communication Cycle in Conjunction with the phases of the protocol


Static Segment
Dynamic Segment
Failure Detection
Membership Communication
Membership Decision
Heartbeat Msgs in Static Slots
Node 1 Membership Opinion
Node n Membership Opinion
General Purpose Dynamic Messages
Network Idle Time

Failure Detection

Heartbeat messages are observed on all active nodes during the failure detection phase. The local opinion comprises of an incoming heartbeat message in combination with a heartbeat status flag (HB flag).

The local opinion concerning a process can be a 'member', which implies that the heartbeat message was received correctly and the HB flag is set to 'beating' while 'nonmember', which implies that the heartbeat message was received correctly but the HB flag is set to 'dead' or the heartbeat is corrupt (content failure) or the slot is silent (silent failure).

Membership Communication

The nodes in the network will perform the membership communication phase during the dynamic communication segment of the cycle. During every cycle, protocol messages, consisting of the local opinions are broadcast during the membership communication phase in the network. Protocol message are allocated priorities, to get precedence over other messages, in every cycle. Each node will form what is called an opinion matrix, which is assembled from the received local opinions in the protocol messages. Each row will contain the opinion from a particular node while columns will contain all nodes opinion concerning a particular process. This allows all nodes will have had an opportunity to send their opinions, at the end of the dynamic segment. If there are no failures during the cycle, then the opinion matrix will be fully populated with opinions from all the nodes. If there any other event triggered message in the cycle, they will follow the protocol messages. The dynamic segment is assumed to consist of a minimum of N dynamic slots, each for every node.

Membership Decision

A candidate membership view is obtained by applying the majority voting decision function to the opinion matrix, in the membership decision phase. Any process from the membership group, hosted by nodes which failed to send messages or have send messages which differ from the result of the decision function are removed from the candidate view. Thus the membership candidate view is further processed to be made available to distributed tasks in the system. Any node which no longer hosts a process in the membership group or any node which leads to the decision function giving an undefined outcome for any process is deactivated. Thus the possible status of any process is: member or non-member.

During the operation of the protocol, it is possible for a node to signal failure. The node which receives the failure signal uses it to modify the membership view bypassing the membership protocol. If the current cycle is affected by further failures, the protocol is executed as usual. Thus the signaled failure mode enables status of the membership view of the system to be reached faster and incurring less overhead by using the protocol itself.

The correctness of the protocol is decided primarily by the fact that half the nodes in the communication network are active during two cycles of the protocol. Each active node send local its local opinion and the decision function masks any faulty opinions (including content failure and silent failure).Thus two cycles of membership processing are required to correctly handle content failures, depending the instance at which failure occurs during the cycle .The faulty node must intelligently detect a failure has occurred and cease membership operation by listening to the opinions of other nodes during the two cycles. The active nodes receive a confirmation as to the failure suffered by the faulty node. A consistent membership status is reached after the two worst cycles after the last failure.

3.3.2 Process Integration

The basic protocol is modified to allow integration of excluded processes and reactivation of deactivated nodes. The reactivation of a node is indirect, since a node is reactivated when the associated process is re-integrated into the network. The heartbeat message sent in the static slot during the failure detection phase is appended with the join request flag. When the run time environment allows an excluded process to integrate m the J flag in the heartbeat message is set. At this instant, the HB flag is set to 'dead', indicating that process is not yet considered to be a member. If a node is reactivated, its membership view is reset, such that all processes in the system are members. This is followed by failure detection and the re-activated node builds its local opinion, participating in the membership communication and decision phases as if an active node and the corresponding process as a member.

The opinion from an active node with a re-integrating process is considered as 'non-faulty', on par with the local opinion of other active nodes, since its opinion is now in accordance with the other active nodes. However, this opinion is not considered by the decision function of the other active nodes. Its opinion is instead checked so that at least the same processes are members, compared to the result of the decision function in the correct nodes. It the opinion is not checked, the reactivating node has suffered failure during re-integration. The node is deactivated and the membership of the processes is removed from the correct nodes. If no such failure occurs then the re-integrating process will rest the J flag and set the HB flag to beating in the next cycle, indicating that the process is considered to be a member and the hosting node is active.

Each node also sends an upper bound on the number of active nodes, in addition to its local opinion. This helps a reactivating node to correctly perform the decision function, though it is reactivated into a system of active and inactive nodes, thereby receiving fewer opinions. The upper bound on the number of active nodes is activated at the end of the membership decision phase, when the number of active nodes in the system is known appropriately. It may be need that the delay from the join request to potential acceptance in the members group is at worst two cycles.

3.3.3 Event Triggered Scheduling

When a change of status occurs i.e. when a heartbeat message is missed or in the event of a re-integration attempt, this step is to perform the membership communication and decision phases .A node may use the membership request flag, in addition to the heartbeat message to communicate to the other active nodes in the system, whether it will perform membership communication or membership decision in the next cycle. When a change is detected in the system or a node receives a heartbeat message with the MR flag set, the node will set it own MR flag to signal to the other nodes. Thus the correct nodes perform membership communication and decision phases. This may occur if a failure affects the last heartbeat message in the cycle. The remaining nodes detect the failure in the next node and the membership phase will be performed .The membership phases are executed in the same cycle as would be performed in the event of change in the membership status. This allows handling of failures from the previous cycle in the membership phase.

A node must also keep track of the "generation number of the membership group, included together with the protocol messages sent in the membership communication. The local generation number is incremented, for every membership phase performed .Thus it may be noted that the generation number of all the nodes must be the same during normal operation. If a node receives an opinion with a generation number higher than its own, it will deactivate itself. Similarly, if a node receives an opinion with a generation number lower than its own, then the opinion from the node is disregarded in forming the opinion matrix.

3.3.4 Decision Function

A decision function returns a consistent decision of the status of a process based on the local opinions of active nodes in the system. Each process in the system can have a differently configured decision function

A decision function is based on two operands, a vector containing local opinions from all the active nodes in the system concerning a specific process and the threshold value 'u' of the number of active nodes in the system. The vector will contain the status of the process, as determined by all the active nodes in the system. Each opinion in the vector may or may not be a member in the system. The threshold value sets the required number of opinions to set up an opinion matrix, since a majority [u/2] can be determined and the outcome of the opinion matrix be defined. The decision function returns the opinion that at least [u/2] nodes have about a process; i.e., member or non member. A lost opinion message indicates a missing local opinion from an active node (or fewer number of active nodes as compared t the threshold value 'u') or the outcome is undefined. The protocol then deactivates the node associated with the missing local opinion. If the number of active and inactive nodes is equal, the outcome is prioritized according to non-member and member nodes.

CHAPTER 4 - Verification Model

In this model we focus on the operation of membership service for the FlexRay Protocol. We assume that the clocks are synchronized in this model.

4.1 Model Layout

The model consists of five components, each with its own template: GlobalClock, Node, Scheduler, DTask and Bus. Each node needs an instantiation of the Node and DTask template. The latter handles processes that run on the node. The former represents the runtime environment microprocessor and handles everything else, like communication with the bus and sending data to the bus.

In the model, for every process running in the node we instantiate DTask template specifying its id, slot number and node (where it is being executed). A process can have membership status: member or not member, initially the membership status of all processes is member. Each DTask i.e. process can only communicate with its own Node and each Node only with its own DTask, not with other DTasks.

The Bus template needs to be instantiated only once, as there is just one Bus to connect all Nodes in the system. This automaton transfers messages from and to the nodes. The GlobalClock automaton is only instantiated once as well. It makes sure the model's cycle completion is known to all automatons.

The FlexRay protocol consists of Static and Dynamic segments followed by Network Idle time; similarly the Node's behavior is modeled in Uppaal. The Scheduler keeps track of time and slot counter for Static and Dynamic segments. The Node's runtime environment communicates with its respective processes and sends the data on the bus when it is scheduled to send during the static segment. Node informs the process about its membership status after every membership communication phase.

In the model, we allow a predefined number 'Slots' for the static segment and the dynamic segment. This is done by manipulating variables 'gStSlots' and 'gMinSlots'. Each static and dynamic slot has predefined time assigned to it. The Scheduler automaton increases the static slot counter by one, whenever the clock reaches the predefined slot time. Scheduler changes its state from Static or Dynamic, whenever clock time reaches ‘gStSlots’ or ‘gMinSlots’. Then the slot counter is reset to 0 and the procedure starts anew.

Since we have assumed that the clock is synchronized, the slot counter is global for this model. ‘vSlotCounter’ for Static Segment and ‘dvSlotCounter’ for Dynamic Segment are the slot counters for each segment. It is assumed that Dynamic Segment schedules all the nodes for membership communication.

Here is an example system definition using three 4 nodes and 5 processes:

```
system Process, N1, N2,N3,N4,c1,d1,d2,d3,d4,d5,b1,a;
```

And the matching process assignments:

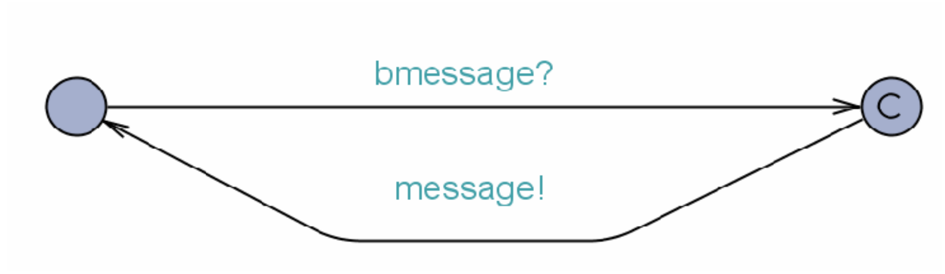
```
N1 = Node (1, 7, 1);
N2= Node (2, 6, 2);
N3= Node (4, 6, 3);
N4= Node (3, 6, 4);

d1 = DTask (1, 1, 1);
d2 = DTask (2, 2, 2);
d3 = DTask (3, 3, 1);
d4 = DTask (4, 4, 3);
d5 = DTask (5, 5, 4);
```

4.2 Bus

First we take a brief look at the Bus. Throughout this model the Bus is treated as a medium, in which a node can put a message and from which the other nodes can then take that message out again. The bus uses two synchronization channels, ‘bmessage’ and ‘message’. Several global variables are used for reading from and writing to the bus. Five of them input data into the bus: 'pID', 'Hb', 'JR', 'M_R', 'NodeID'. This sets the message the bus will send. 'NodeID' should be the sending node's id. 'Hb', 'JR', 'M_R' are the node's heartbeat status. 'pID' is process ID sending the message along with its heartbeat.

Figure 4.1 shows the automaton of bus. Channel 'bmessage' is used for receiving message from the nodes which is the broadcasted to other nodes in the system.

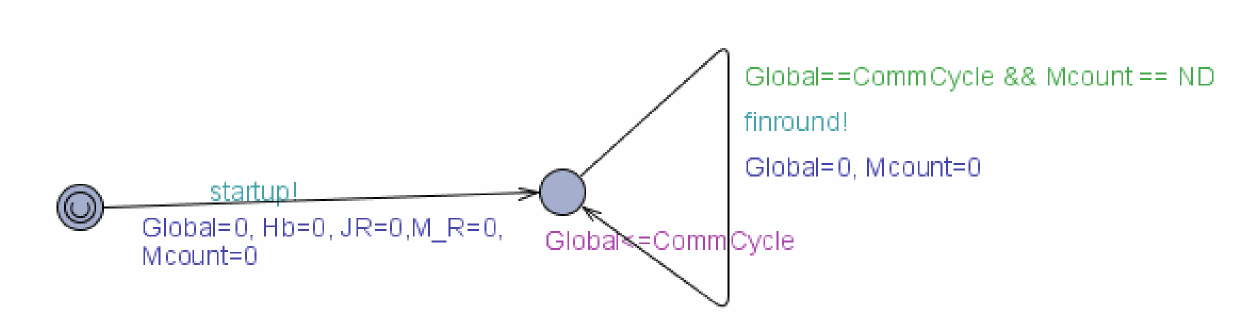
Figure 4.1 Bus Model in the System



4.3 Global Clock

The automaton synchronizes with other automaton using the channel 'startup', which initializes all the system. GlobalClock indicates that a round of communication is finished after every 'CommCycle' time and synchronizes with channel 'finround' to let other automaton know that a round of communication is over.

Figure 4.2 Global Clock

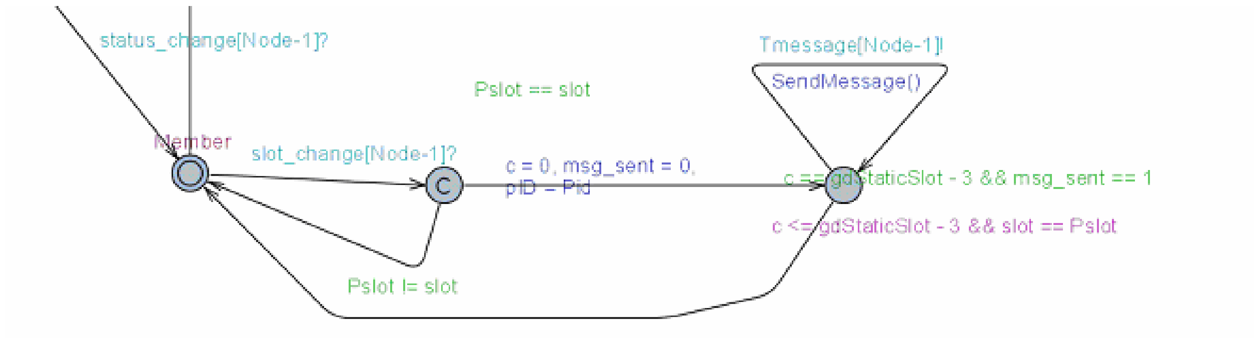

Global==CommCycle && Mcount == ND
finround!
Global=0, Mcount=0
startup!
Global=0, Hb=0, JR=0,M_R=0,
Mcount=0
Global<=CommCycle

4.4 DTask

The DTask template has number of parameters to separate any number dtasks; i.e., processes. They are: ‘Pid’, ’Pslot’ and ‘Node’.

‘Pid’ is the process id specific to the process and is used to indicate process which is sending the data on the bus. ‘Pslot’ is the slot number in which the process is scheduled to send data. ‘Node’ is the Node ID in which the process is running and is used for sending data to Node automaton using synchronization channel ‘Tmessage [Node]’.

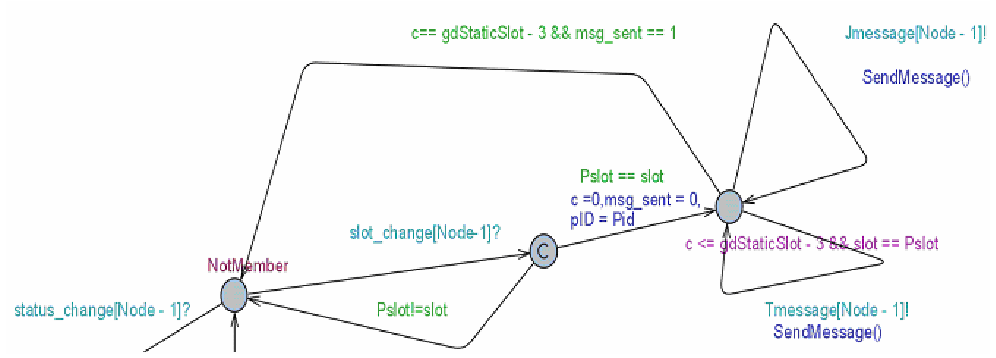
Figure 4.3 Processes in Member State



As shown in figure 4.3. initially Dtask automaton is at ‘Member’ state, since all processes are members of the membership group. Whenever there is change in slot, current slot is checked with ‘Pslot’ to check if it can send data now. When ‘Pslot’ is equal to the current slot in the system, process sends data to the run time environment through Tmessage channel.

DTask automaton makes a transition to ‘NotMember’ state, when the Node automaton synchronizes through the channel ‘status_change [Node]’. When the DTask is in ‘NotMember’ state and scheduled to send data on bus. It sends a join message request using the ‘Jmessage [Node]’ channel, see fig 4.4.

Figure 4.4 Processes in NotMember State

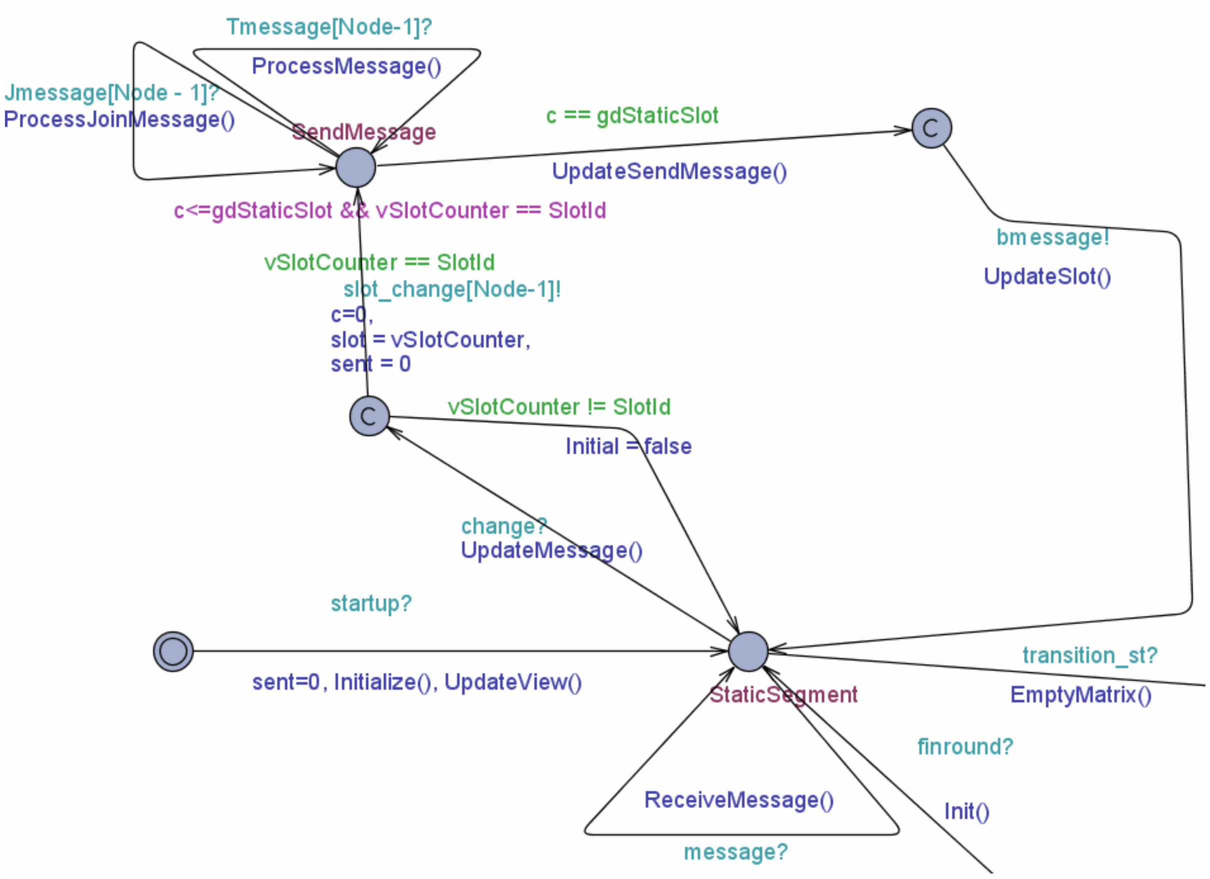


4.5 Node

The Node template has a number of parameters to allow a number of nodes to operate in the system. These parameters are: dSlotId, time, Node

‘Node’ ties this Node to a specific Node automaton and is used to share data between a Node and its processes. No other automata should access that data, although it would technically be possible to do that in UPPAAL.

Figure 4.5 Static Segment



As shown in fig 4.5, ‘startup’ is the synchronization channel to initiate a Node. Once start up is completed, a node is in Static Segment state from there it can take three transitions. First, when the slot counter is increased, slot which is scheduled to send data on the bus. This happens through synchronization of ‘change’ channel. Afterward the Node automaton checks if the current slot of the system scheduled is equal to the slot number in which it is supposed to send. If the current slot in not equal to its slot number it returns to its previous state – StaticSegment.

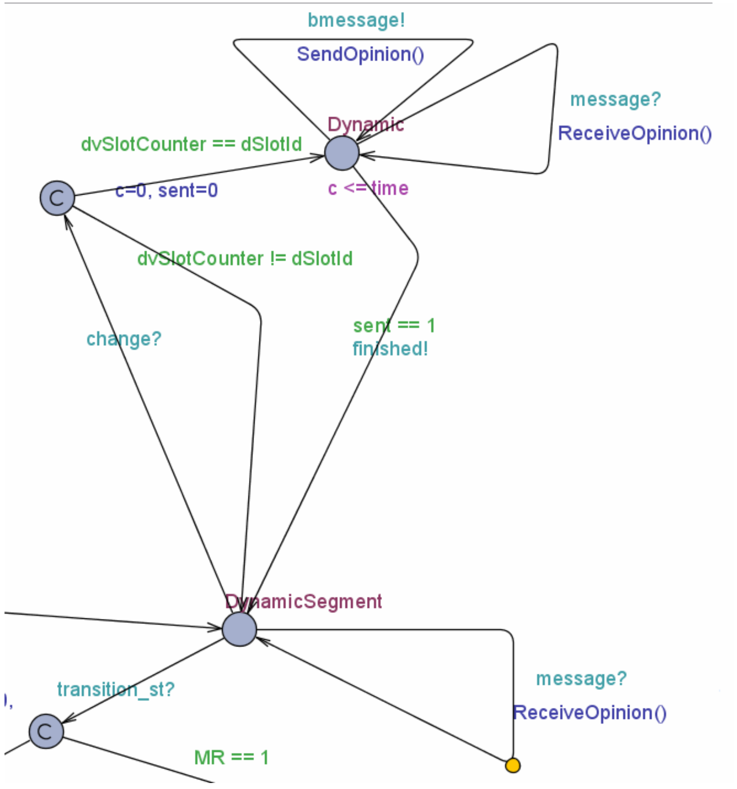
The second transition is taken by synchronizing with the ‘message’ channel, through which the node receives data sent by other nodes. The third transition is taken when the node synchronizes with ‘transition_st’ to go to dynamic segment.

The Scheduler automaton will regularly update the slotcounter variable, as we will see in the next section.

Once the Node verifies that it is scheduled to send data on the bus, it moves to 'SendMessage' state. During its transition to 'SendMessage' state, Node indicates to the process which is scheduled to send data which we have explained in previous section.

The Node automaton receives either a message with heart beat or a message with a join request using synchronization channels 'Tmessage [Node]' and Jmessage [Node] respectively. Once the Node receives message from the process (DTask), it processes the message to be sent on the network. The runtime environment checks for the current view of the process and updates the status flags accordingly. The message is then sent on the bus using the synchronization channel 'bmessage'.

Figure 4.6 Dynamic Segment in Node



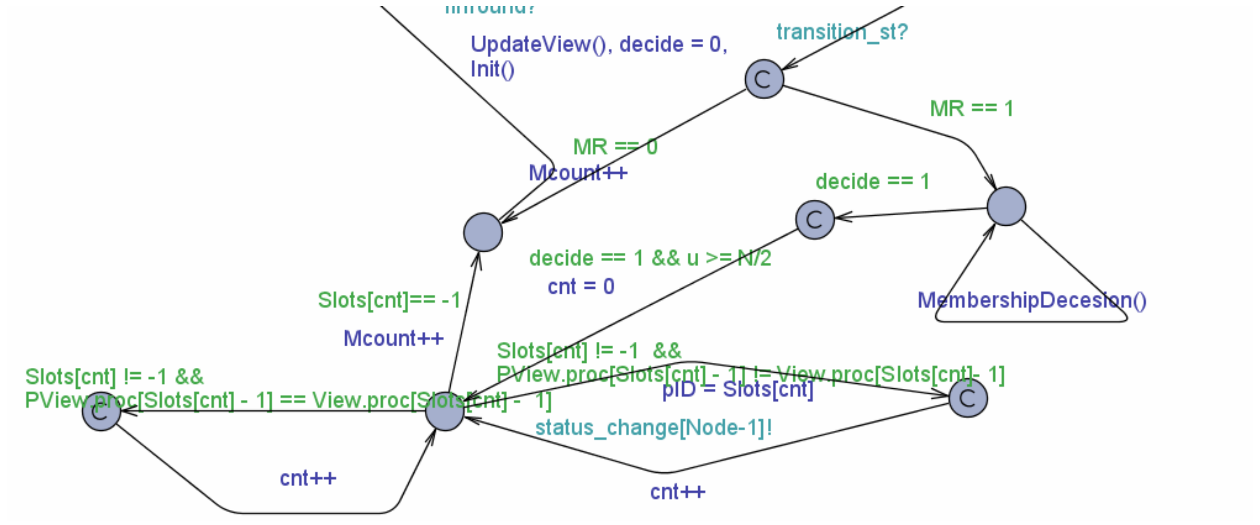
Now we will talk about the 'DynamicSegment' state where the Node takes the transition after 'gdStaticSlot' time. The opinion matrix is first initialized to empty. In this state the automaton can take three transitions. First, if there is change in dynamic slot which is scheduled to send data on the bus. This happens through synchronization of the 'change' Channel. Afterward the Node automaton checks if the current dynamic slot of the system scheduled is equal to the slot number in which it is supposed to send. If the current slot in not equal to its slot number it returns to its previous state –DynamicSegment.

The second transition is taken by synchronizing with the ‘message’ channel, through which the node receives data sent by other nodes. The opinions sent by other nodes in system are recorded in the opinion matrix. The third transition is taken when the node synchronizes to go to Network Idle state.

In the State dynamic, the Node send its opinion on the bus when membership communication request has been placed by another node or when the node has detected a change in its opinion during the StaticSegment state or Failure detection phase.

Once it has sent its opinion it informs the scheduler automaton that it has finished sending on the bus through the finished synchronization channel.

Figure 4.7 Network Idle State in Node

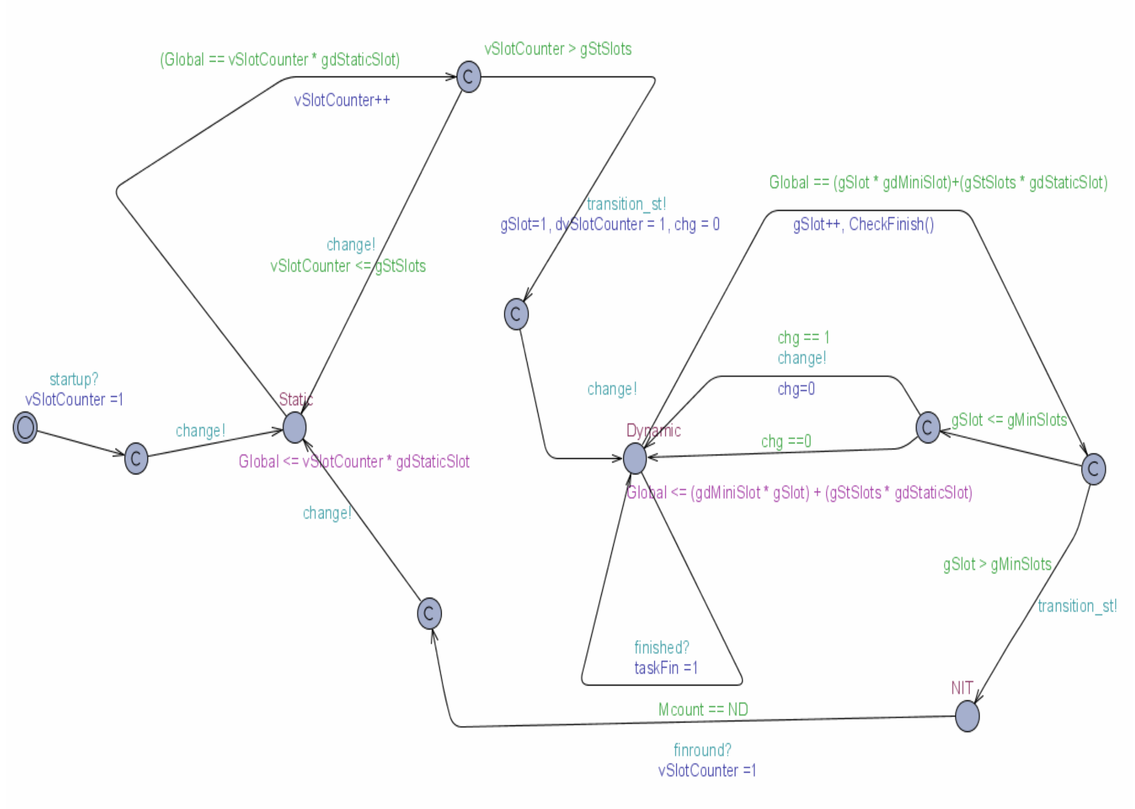


During the Network Idle Time, the Node performs membership decision function. Once the new view of the processes in the system is updated, all nodes update the status of the process.

As shown in fig 4.7, array Slots contains the Pid of all processes running on Node. For each process in the node, we check if there is change in opinion about the process. If yes then the process is assigned status NotMember by synchronizing with channel “status_change”.

4.6 Scheduler

Figure 4.8 Scheduler



The Scheduler automaton, shown in fig 4.8, is responsible for maintaining slot counter for Static and Dynamic Segments.

On Startup, the slot counter for Static Segment ‘vSlotCounter’ is initialized to 1. The scheduler increments the slots counter after every ‘gdStaticSlot’ time. When the slot number is changed, all the nodes know about it through the synchronization channel ‘channel’. When the ‘vSlotCounter’ reaches the maximum number of Static slots to be scheduled in Static segment, it transition to ‘Dynamic’ state and the nodes also transition to Dynamic Segment. This is done through the synchronization channel ‘transition_st’.

In ‘Dynamic’ state, the scheduler automaton keeps track of the dynamic mini slots and the dynamic slot. A Dynamic Slot is made up of dynamic mini slots. The Scheduler increments the mini slots counter after every ‘gdMiniSlot’ time. When the Node automaton synchronizes with channel ‘finished’, indicating that it has finished using the bus, the Dynamic slot counter ‘dvSlotCounter’ in CheckFinish() is incremented. This change in Dynamic slot counter is informed to the Nodes using the channel ‘change’.

Once the maximum number of mini slot is reached it transitions to NIT state. The Scheduler automaton stays in NIT state until it synchronizes with channel ‘finround’ with Global Clock.

CHAPTER 5 - Verification of Model

The model that we have constructed is checked by certain number of behavioral properties. Behavioral properties extract the specification of the system. We need to see that the model respect the same behavioral constraints as its specifications. The list of such properties is given in table.

Table 5.1 Verification Properties

Properties	Verified
A[] not deadlock The system must have a cyclic infinite behavior, and never be deadlock	Yes
E<> N1.State ==-1 In normal execution (without fault), *state* must never be equal to -1	No
A [] not (N1.SendMessage and (N2. SendMessage or N3. SendMessage or N4. SendMessage)) Only one node can be in SendMessage state at the same time	Yes
E<> d2.NotMember In an environment with faults, verify that Process 2 can be removed from the group by protocol.	Yes
E<> JR = 1 and d2.Member A join request can never be made when the process is member of the group.	Yes

5.2 Liveness Property

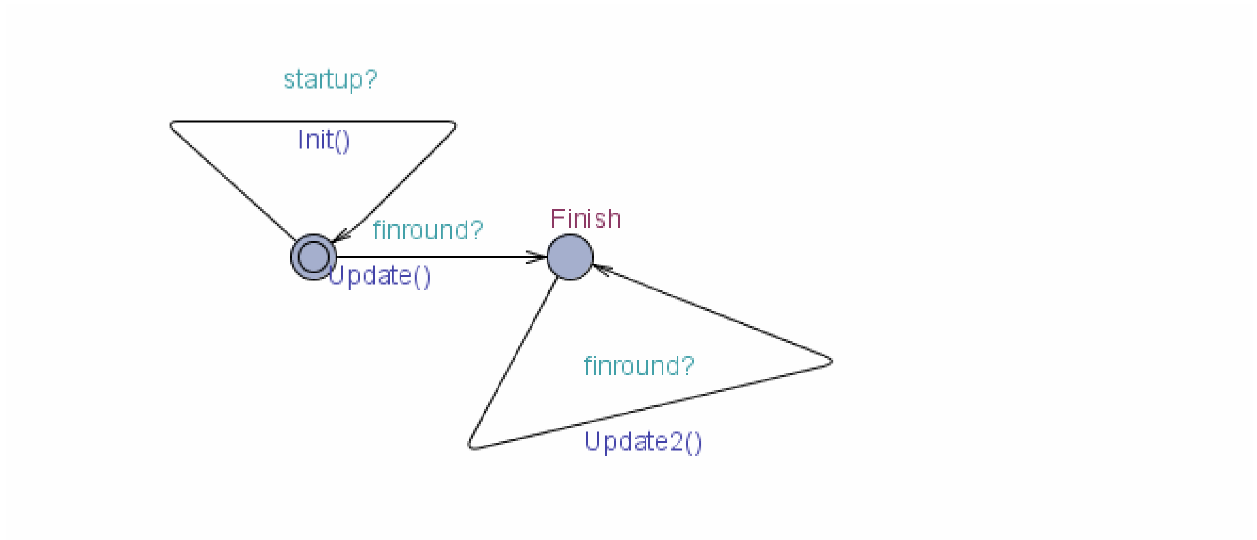
We use an Automaton ‘Agreement’ to check the liveness property of the model. In this we maintain an observer view on the membership of each node; hence we use this if the membership view of all the nodes is same as the observer view. We check:

c1.Static --> a.Agree ()

Our model satisfies this property.

The automaton for checking the property is show in figure 5.1

Figure 5.1 Agreement Automata



The code for Agree is as follows

```
bool Agree () {
   int i, j;
   bool status;
   status = true;
   for (j =0; j<ND; j++) {
      for (i=0; i<N; i++) {
                 if (oView.proc[i]! = aView[j].proc[i])
                      status = false;
           }
      }

    return status;
}
```

CHAPTER 6 - Analysis

The performance of the membership protocols was measured using fault injection techniques. We did a test to perform the response time on nodes. A test was performed to check the amount of failure the protocol can handle for different number of nodes over period of time.

6.1 Response Time

Figure 6.1 shows how the response time follows the cycle time. The test includes three test cases, best case, median case and worst case. The best case is measured by injecting a fault in message slot 6, the last sent static message. The median case is injected in slot 4 and the worst case is injected in slot 1. Figure 6.2 illustrates how the response time changes relatively to the cycle time by showing how many percent of the cycle time it took for the protocol to detect a change in the cluster.

Figure 6.1 Absolute Response time

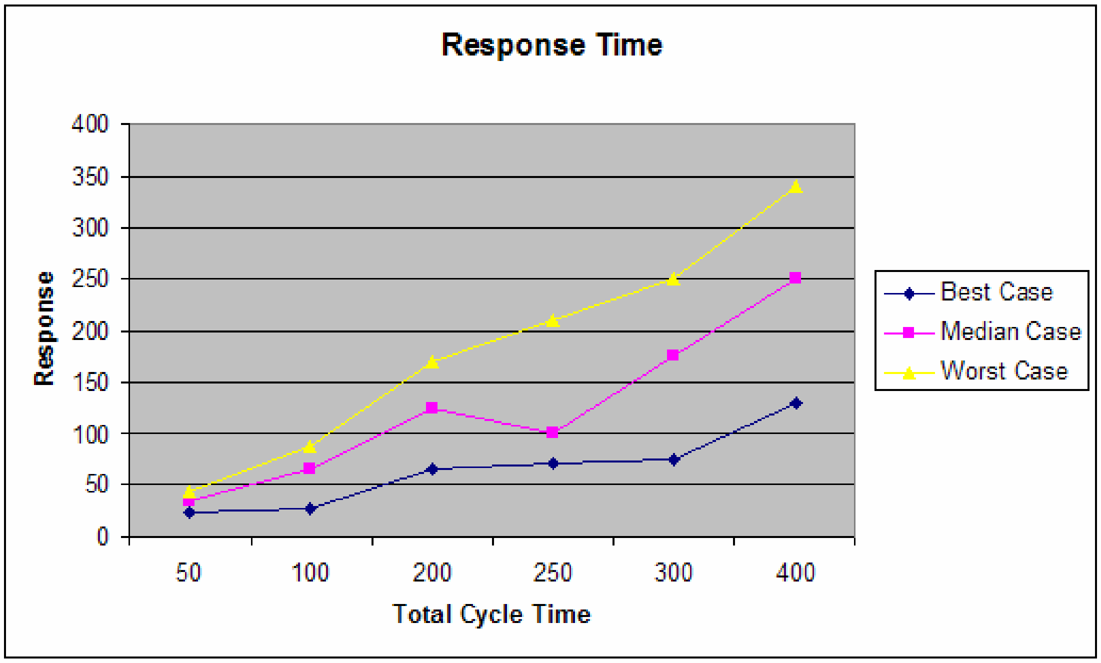
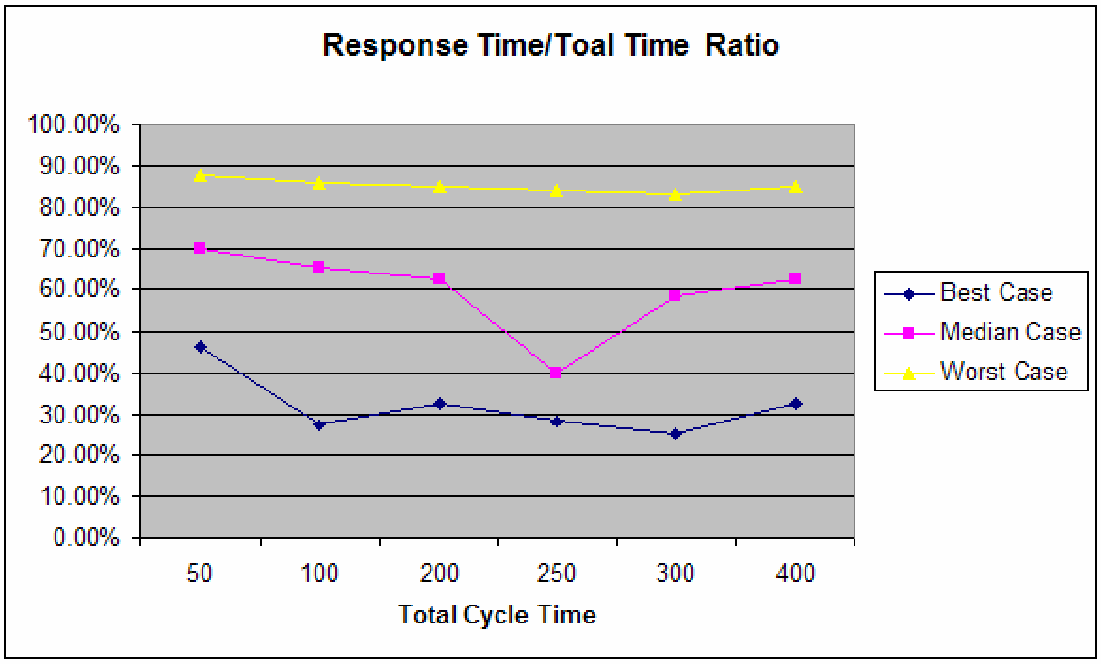


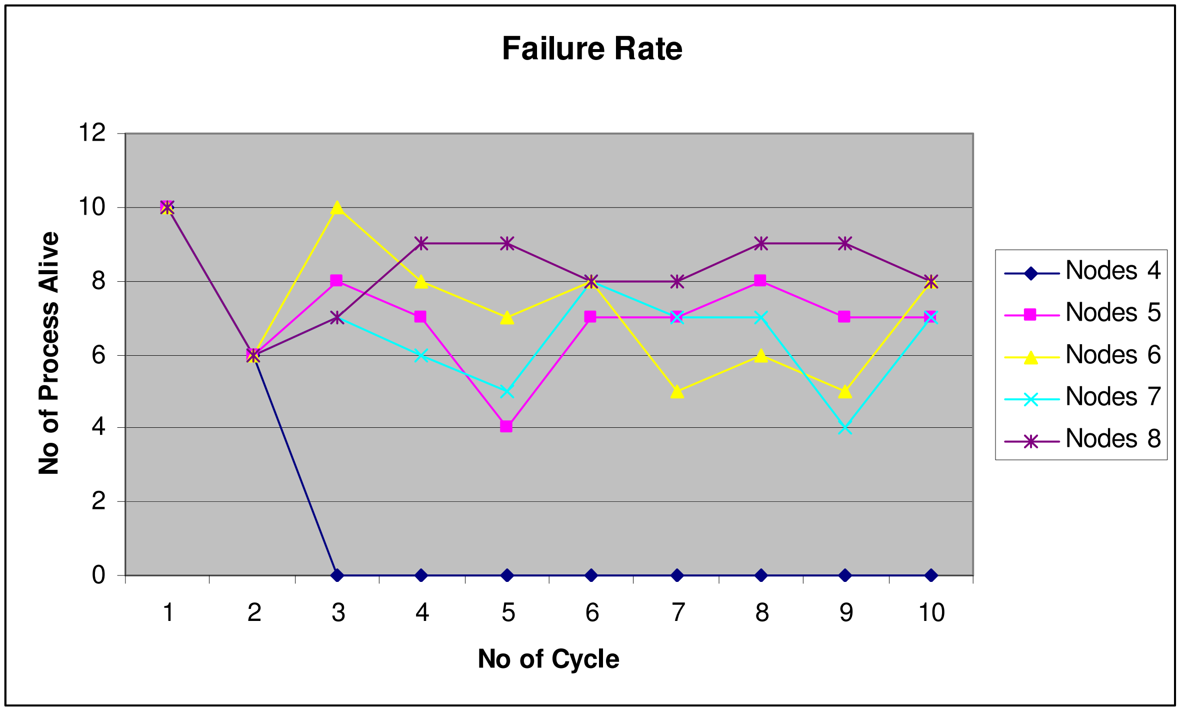
Figure 6.2 Relative Response time



6.2 Failure Rate

We conducted test where there is a fault in the system in every cycle. This fault is an asymmetric fault caused by a process in Node 'N4'. The message sent by the node is detected to be corrupt by one node in the system. The message could be corrupted due to disturbances in the network. The node changes its opinion about the process and signals for a membership communication phase to other nodes. After membership communication phase node which reported suspicion about the process is ruled out to be faulty and removed from the group. Thus the processes which are running on the node are also not a member of the group.

Figure 6.3 Failure Rate for Different number of Nodes



In the above test, we have considered 10 processes running on different number of nodes. The processes are not distributed uniformly among the nodes in the system. We can see that a system with 8 nodes would handle the faults optimally, but our main aim is to reduce to number of nodes in the system. From figure 6.3 we can see that a system with 5 nodes tries to match the performance that we have with 8 nodes. Thus for a system with 10 processes it will be more cost effective to have 5 nodes. Hence we can have a system with $N > 3a + 2s$, where 'a' is asymmetric faults and's' is symmetric faults. In classic Byzantine models, $N > 3t$, where t is the number of faults in the system.

6.3 Node Failures

From Section 6.2 we can see that Nodes that have single opinion differing from other nodes is removed from the group. This causes the processes running on the node to be out of the group. This causes overhead on the system for reintegration of the processes. We can avoid this situation by providing an alternative approach while making decision. If there is a Node having a result differing from other nodes in the system, we need to find how many such results differ. If the count is more than u/2 then we should remove the node from the group. If it is less than u/2 it should be kept in the group. The node should correct its View about the process according to the opinions of other nodes.

6.4 Verification Results

Figure 6.4 Verification Time

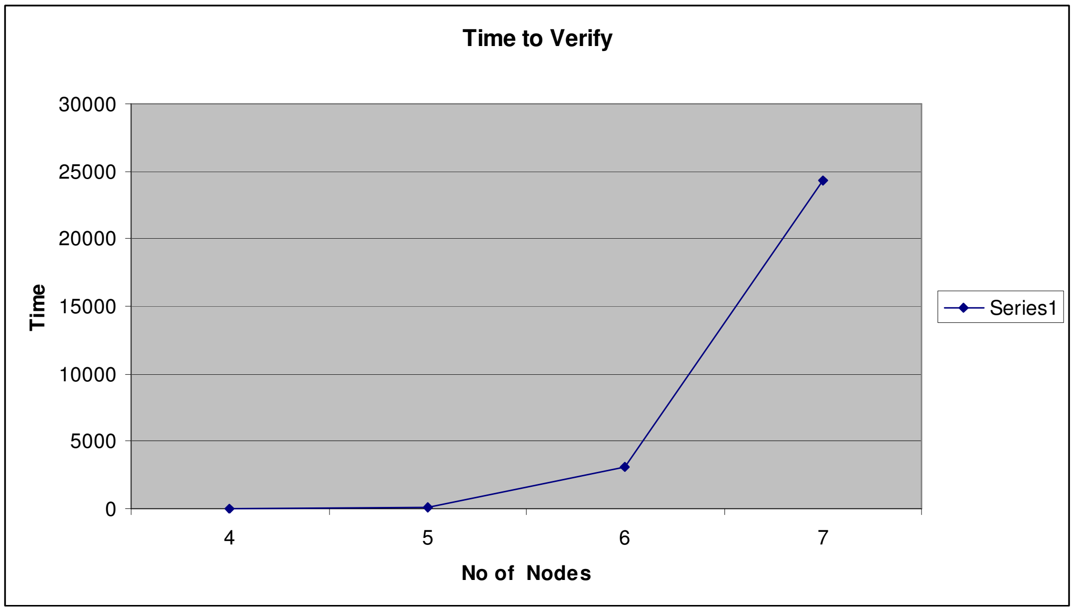
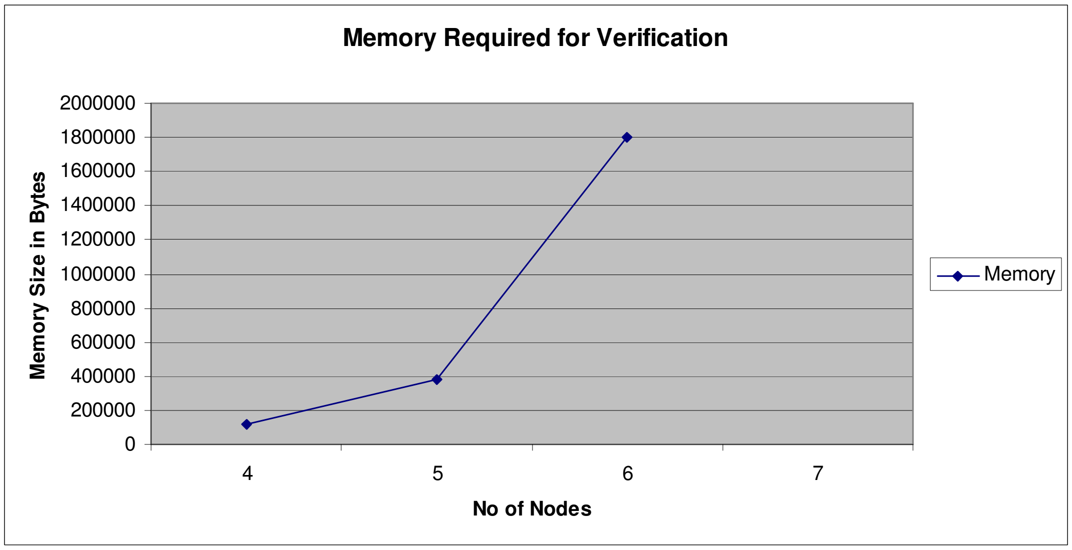


Figure 6.5 Memory Required for Verification



We verified the Agreement property for configurations with four, five, six and seven nodes. Figure 6.4 shows the time taken in checking the property presented in section 5.2. For the case of 7 nodes, it leads to exhaustion of memory resources. The figure 6.5 clearly show that the state space size increases exponentially with the number of nodes in the system.

Nevertheless, to be able to check the membership protocol for 6 nodes gives us a high confidence level in its correctness, because with 6 nodes we are able to generate rather subtle fault scenarios, such as the masked faults, that arise with the simultaneous fault of two nodes, which may be either members of the group or attempting to join. Although checking the correctness of the membership protocol for 8 nodes would provide an even higher confidence, because with that many nodes we could consider scenarios with 3 simultaneous faults.

CHAPTER 7 - Conclusion

7.1 Conclusion

In this thesis we have examined an existing membership protocol for a FlexRay system. The protocol was modeled on UPPAAL to check for the correctness of the protocol, using properties to check if the model meets all requirements. The results obtained show that the membership protocol satisfies its specification for configurations of up to 6 nodes, providing us further assurance on its correctness. We believe that the abstraction applied to this model is not far from the actual membership protocol.

7.2 Future Work

The future work need to be done is to compare the model with the TTA validation model [13]. We will be able to determine how both TTP and FlexRay work with their membership protocol.

References

[1] Carl Bergenhem, Johan Karlsson "A Process Group Membership Service for Active Safety Systems using TT/ET Communication Scheduling", Presented at 13th IEEE International Symposium on Pacific Rim Dependable Computing,Vol. 00, pp 282-289, Dec 2007.

[2] M. Ayoubi, T. Demmeler, H. Leffler, and P. K¨ohn, "X-by-wire functionality, performance and infrastructure", Presented at the Convergence Conf. 2004, Detroit, MI.

[3] Neil Storey, Safety-Critical Computer Systems, Addison Wesley Longman, 1996

[4] FlexRay – http://www.flexray.com

[5] Fujitsu – http://www.fujitsu.com/global/services/microelectronics/technical/flexray/index_p11.html

[6] N. Suri et l.,"Reliability Modeling of Large Fault-Tolerant Systems", Proc. FTCS-22, pp. 212-220, 1992.

[7] N.Suri, C.Waler, and M.Hugue, "Advances in Ultra-Dependable Distributed Systems", IEEE CS Pres, 1995.

[8] L.Lamport and P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults", J. ACM, vol 32, no. 1, pp. 52-78, Jan 1985.

[9] http://www.ixxat.com/introduction_flexray_en.html

[10] FlexRay Protocol Specification v2.1, Dec 2005.

[11] Gerd Behrmann, Alexandre David, and Kim Larsen, "A Tutorial on UPPAAL", Nov 17 2004.

[12] Günter Reichart, "FlexRay Enabler for Future Automotive System Architectures", Mar 2005.

[13] Karen Godary, Isabelle Auge-Blum, Anne Mignotte, "Temporal bounds for TTA: Validation", DIPES, pp 73-82, 2004.