Formal Verification of Fault-Tolerant Startup Algorithms for Time-Triggered Architectures: A Survey*

Indranil Saha, Suman Roy and S. Ramesh

***Abstract*—Time-triggered architectures form the important component of many distributed computing platforms for safety-critical real-time applications such as avionics and automotive control systems. TTA, FlexRay and TTCAN are examples of such time-triggered architectures that have been popular in recent times. These architectures involve a number of algorithms for synchronizing a set of distributed computing nodes for meaningful exchange of data among themselves. The algorithms include a startup algorithm whose job is to integrate one or more nodes into the group of communicating nodes. The startup algorithm runs on every node when the system is powered up, and again after a failure occurs. Some critical issues need to be considered in the design of the startup algorithms, for example, the algorithms should be robust under reasonable assumptions of failures of nodes and channels. The safety-critical nature of the applications where these algorithms are used demand rigorous verification of these algorithms and there have been numerous attempts to use formal verification techniques for this purpose. This paper focuses on various formal verification efforts carried out for ensuring the correctness of the startup algorithms. In particular, the verification of different startup algorithms used in three time-triggered architectures, TTA, FlexRay and TTCAN, are studied, compared and contrasted. Besides presenting the various verification approaches for these algorithms, the gaps and possible improvements on the verification efforts are also indicated.**

***Index Terms*—Time-Triggered Architectures, TTA, FlexRay, TTCAN, Startup Algorithms, Fault Tolerance, Modeling, Verification**

I. Introduction

Mission-critical Cyber-Physical System (CPS) applications in areas such as transportation (automotive, aerospace, and railways), industrial automation and process control, and medical systems rely heavily on dependable embedded systems [1], [2]. These applications try to ensure meeting hard real-time requirements in a dependable and predictable manner to guarantee users' physical safety. Until recently, the practice was to hand-craft these systems in an ad hoc manner. Time-triggered architectures (TTAs) are seen as a solution to this problem [3]. In this framework, independently developed functions can be integrated with a minimal integration effort, thereby offering strong composability. They give effective barriers to fault propagation, and implement fault-tolerance by the active replication of components.

Time-triggered architectures provide support for synchronizing multiple computing nodes to perform different tasks and exchange messages, in order to cooperate among themselves. The nodes in such distributed networks share one or more communication buses for exchanging messages. The nodes access each bus following a Time Division Multiple Access (TDMA) scheme. At pre-specified instants of time, a node writes messages on the bus, and all the other nodes connected to the bus read the messages. All system activities are triggered by the progression of time [4] and the computing nodes are required to synchronize their clocks to share a common notion of time; clock synchronization is a challenge as the local clocks may drift in time. The task schedules in different nodes and the message schedules are decided a priori at design time. Each node is equipped with a communication controller which knows when to send a particular message, and when to expect a message from another node.

For proper operational purposes, time-triggered architectures depend on a variety of basic algorithms: distributed clock synchronization, startup/reset, bus guardian window timing, group membership, clique avoidance, non-blocking write [3], to name a few. As in steady-state operation the nodes of a time-triggered architecture use the TDMA strategy to access the bus, these architectures require a startup algorithm to reach a steady state operation mode after power on, or after recovery from a failure. In the steady state, the bus access time is divided into a cyclic sequence of slots with each slot assigned to a node for sending its message. Each node has a mapping from its local time to a slot in the cyclic TDMA schedule; in particular, each node knows the local time at which the slot for another node begins. The operation of a time-triggered architecture depends on the synchronization of the local clocks, which simply means that any two non-faulty nodes agree on slot delimitation at any instant of time. Thus, the synchronization problem involves adjusting the values of local clocks such that the nodes continue to remain synchronized even if hardware clocks make a drift (due to a small differential in the rate of their oscillators). The startup problem is to establish values for the local clocks as the nodes power up in the beginning to become quickly synchronized; the restart problem is to establish synchronization once more after the transient faults have afflicted the values of the local

*This work was partially carried out as part of a project under an MoU between Honeywell Technology Solutions Lab, Bangalore and GM India Science Lab, Bangalore

Indranil Saha is with the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur, India. e-mail: isaha@cse.iitk.ac.in

Suman Roy is with Infosys Ltd., Bangalore, India. e-mail: suman_roy@infosys.com

S. Ramesh is with General Motors Global R&D, Warren, MI, USA. e-mail: ramesh.s@gm.com

Manuscript received May 30, 2015; revised September 30, 2015.

clocks at one or more (or all) nodes.

The correctness and reliability of time-triggered systems depend on the correctness of these algorithms, besides that of the applications running on these platforms. Many applications that use time-triggered architecture are safety-critical, and hence demand rigorous verification of these algorithms. As these algorithms contain complex interactions between the components of the system, it is extremely hard to confirm their correctness through manual analysis or through simulation based on a limited number of test cases. Automated formal verification tools and techniques come here to our rescue by providing algorithmic mechanisms to explore the state space of the system systematically and guaranteeing that the algorithms are bug-free. There have been a number of attempts to use formal verification techniques for verifying these algorithms. Among all the algorithms, startup algorithms have attracted the attention of many researchers [5], [6], [7], [8], [9]. The startup algorithm is a complex distributed algorithm involving the interaction of system components in interesting ways during the startup process [10], and this prompted us to make a comprehensive review of the literature on formal verification attempts of these algorithms.

Contribution of the Paper

In this paper, we survey the verification approaches of the startup algorithms employed in different time-triggered architectures. We focus mainly on three time-triggered architectures: TTA, FlexRay and TTCAN, which are being widely adopted in automotive and aerospace industries. Though a number of algorithms are used in time-triggered architectures, we concentrate only on the startup algorithm in particular as verification of the startup algorithm has received the utmost attention from the formal methods research community. Our objective is to present all the relevant aspects of verification of an algorithm for a time-triggered architecture by considering the startup algorithm as an example.

There are two aspects of startup algorithms that need careful modeling: time and faults. Moreover, the properties of interest have to be identified and a suitable tool has to be chosen for automatic verification. Although existing work in the literature select different modeling techniques, properties and tools to suit their purposes, there has been no effort so far to consolidate the different modeling and verification efforts. In this paper we deal with different ways of modeling time, and different fault hypotheses that are considered while designing fault-tolerant algorithms for time-triggered architectures. We list the properties that are of interest for the verification of these algorithms and identify different tools that have the potential to be useful for this purpose. Finally, we produce a summary of the verification efforts of startup algorithms to date, thereby bringing out a comparative analysis of them. This summary reflects some interesting observations, such as finite state modeling does not lead to the degradation of the performance of the model, *e.g.*, scalability of the model with respect to the time required to complete verification. Another interesting observation is that a single fault hypothesis is of interest for modeling faults in most of the work. From this analysis, one can get useful insights into the strengths and weaknesses of different approaches that have been used so far, which modeling approach is well suited for a particular algorithm, and what kind of scalability one should expect from the current tools. We also list the scope for future work in this area. We hope that this work will trigger research efforts for the verification of other important algorithms for time-triggered architectures in the future. Overall, this survey article can act as a useful and important reference for someone interested in modeling and verifying any real-time distributed algorithm in general, particularly for time-triggered architectures.

Organization of the Paper

The rest of this paper is organized as follows. In Section II, we provide required background on time-triggered architectures. In Section III, the startup algorithms employed in three different time-triggered architectures are discussed. In Section IV, we present formal models of faults for these architectures which are assumed during the design of the basic algorithms for them. In Section V, we discuss the different kinds of approaches to model time, and also compare their merits and demerits. The properties relevant to the verification of these startup algorithms are discussed in Section VI. The different verification approaches of the startup algorithms are presented in Section VII. In Section VIII, we discuss the pieces of work that have been carried out to model-check these startup algorithms. Finally, we conclude the paper in Section IX by providing some future directions of research.

II. Time-triggered Architectures

The complexity of control functions in automotive and aerospace systems has increased greatly over the years. This has resulted in the transition of implementation platforms from stand-alone independent subsystems to distributed controllers called Electronic Control Units (ECUs) that exchange local state information among themselves for performing different control functions. Henceforth, an ECU will be referred as a *controller* or as a *node*. The distributed architecture additionally provides a greater degree of fault-tolerance required for many safety-critical applications. The distributed platforms are often bus-centric systems rather than point-to-point networks, reducing their cost and weight. There are two main types of communications that are used in these systems: event-triggered and time-triggered.

A. Event-triggered Communication

In event-triggered systems, messages are exchanged between different controllers when new events are generated locally. State changes are caused by events which, in turn, trigger messages. For example, a driver pressing a brake pedal is sensed by a controller that sends a message to an appropriate controller which computes the necessary response. Event-triggered communication networks utilize resources more efficiently, as messages are sent across only when needed, and can accommodate additional nodes easily. These systems provide a flexible allocation of resources which is quite useful

when demands vary a lot. However, though in safety-critical applications some basic quality of services should be ensured, event-triggered protocols hardly provide any guarantee on the quality of services.

B. Time-triggered Communication

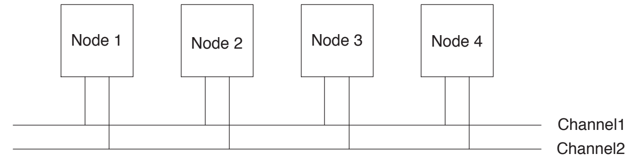
In a time-triggered system, message exchanges take place at statically pre-defined points of time, irrespective of whether there is a state change or not. All the communicating nodes are synchronized in time and every activity on the network is time-stamped using global time. This makes the communication infrastructure in a time-triggered system deterministic and predictable. The contention is resolved during design time when the schedule is constructed. The static schedule of sending messages over the network is programmed into the controllers, which as a result, know when to send a message and when to receive a message from the bus. This schedule helps control the *babbling idiot failure mode* [11]. An independent component, *viz.* a *bus guardian*, is employed to achieve this, and it lets each node transmit on the bus when it is allowed to do so. A guardian knows the global schedule and itself possesses an independent clock, according to which it allows its nodes to broadcast. No source or destination address is associated with the messages as the global schedule activates all communication. Each node knows the address of the nodes to which a message is to be sent, or from which a message is to be received. This results in a reduction of the size of each message, thereby increasing the message bandwidth of the bus.

A time-triggered bus architecture should support fault-tolerant synchronization. Each node maintains a local clock which is made to be synchronized with the clocks of other nodes. The tightness of the bus schedule, and hence the efficiency of the bus, is linked with the preciseness of synchronization achieved. The time-triggered architecture is also designed to tolerate any single physical fault in any of the nodes and channels without causing any loss of functionality of the system. Failure in communication channels is reduced by employing *dual channel* connectivity between communicating nodes. Important data can be replicated in both the channels and sent across so that even if one channel fails, the message can reach its destination through the other channel. Some amount of fault isolation is also built into the system so that if an error occurs, the error is confined within a subsystem and prevented from propagating throughout the network. In subsequent subsections, we will discuss different time-triggered architectures in detail.

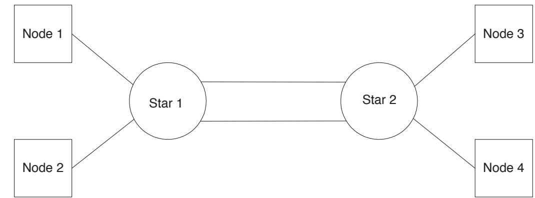
C. Time-triggered Architecture Topologies

The topologies of time-triggered architectures can be of the following three types.

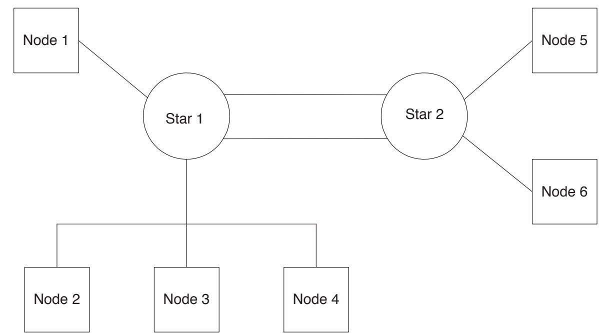
Bus Topology: In a bus topology all the nodes are connected to one or two buses. Bus topology is *passive* as the signal sent by a node is not affected by the channel. An example of a double channel bus topology is shown in Figure 1(a).



(a) Dual Channel Bus Topology



(b) Dual Channel Star Topology



(c) Hybrid Topology

Fig. 1. Time-Triggered Architecture Topologies.

Star Topology: In a star topology, the nodes are connected to each other through star couplers. A network can have at most two star couplers and each coupler accepts an incoming signal, and passes the signal to all nodes connected to the coupler except the sender node. An example of a dual channel star topology is shown in Figure 1(b).

Hybrid Topology: A hybrid topology is a combination of bus and star topologies. Hybrid topologies may be of different types based on how the bus and star topologies are connected to each other through the star couplers. An example of a hybrid topology is shown in Figure 1(c).

D. Examples of Time-triggered Protocols

In this subsection, we provide brief introductions of three time-triggered protocols - TTA, FlexRay and TTCAN.

TTA: The Time-Triggered Architecture (TTA) is proposed as a computing platform for fault-tolerant safety-critical control applications found in the automotive and the avionics domain. The suite of distributed algorithms that are used in this architecture is known as TTP/C and is due to Kopetz and

his colleagues at the Technical University of Vienna [12]. Its current specifications have been developed by TTTech of Vienna [13][1]. In this architecture, an ultra-reliable logical bus connects the host computers that implement the chosen application. Each host computer is connected to the system through a TTA controller. A node consists of a host and its controller. The nodes communicate over redundant channels. The channels may be designed as a physical bus. Alternatively, this architecture can be designed using a "star" topology with a central guardian at the hub of each star. The defense against error propagation is provided by the central guardians. With respect to protocol execution, each central guardian is aware of the parameters of its attached nodes, and can therefore check if a message sent by a node is within its assigned slot and satisfies certain consistency checks. Guardians relay valid messages to all the other nodes on their channel, so that the nodes can view the channel as a broadcast bus.

FlexRay: FlexRay [14] is a communication system and a set of protocols to support the future needs of in-car safety-critical control applications. It is being developed by the FlexRay consortium consisting of many automotive OEMs and suppliers and was founded in 1999. The main objective of the consortium is to develop a flexible and fault-tolerant communication protocol for in-vehicle distributed control systems. A FlexRay system consists of a number of ECUs connected to one or two communication channels through bus interfaces, forming a cluster. The topology of the cluster can be a bus, a star or a hybrid type. An important feature of FlexRay is that it supports both time and event triggered communication.

TTCAN: TTCAN is the time-triggered extension of the CAN protocol [15], [16]. The goal of time-triggered operation on CAN is to guarantee a deterministic communication pattern on the communication bus. A time-triggered and periodic communication takes place in TTCAN, which is clocked by a specific synchronization message sent at regular intervals by a special node, called the *time master*. The synchronization message is called the *reference message*, and is easily recognized by its identifier. The *basic cycle* captures the time period between two consecutive reference messages. There may be several time windows of different sizes in a basic cycle. They provide the necessary space for the transmission of the messages. Both periodic state messages and spontaneous messages can be transmitted in the time windows of a basic cycle. Periodic messages are communicated in *exclusive time windows*. *Arbitrating time windows* are meant for the transmission of spontaneous messages. Bitwise arbitration takes place within an arbitrating time window to decide which message will succeed in writing on the bus. It is also possible that during design time *free time windows* are reserved for further extensions of the network. If new nodes need further space for communication, or the bandwidth needs to be extended for existing nodes, the free windows are suitably converted to exclusive or arbitrating time windows.

[1]Although TTA stands for a generic time-triggered architecture, we label the startup algorithm for the TTP/C architecture as the TTA startup algorithm.

Fig. 2. TDMA round with four slots

III. Startup Algorithms

In the steady state operation of a time-triggered system, the nodes execute a time-triggered communication based on the Time Division Multiple Access (TDMA) cycle as decided by the designer. A TDMA cycle is divided into a number of slots (not necessarily of equal length). A TDMA cycle with four slots is shown in Figure 2. The objective of the startup algorithm is to map the local time of each node to the slots in the TDMA cycle. When the startup process is completed, any node i should know the local time $s_i(k)$ at which the slot for node k begins. After successful startup, for any two non-faulty nodes i and j, the global time instants when i's local clock reads $s_i(k)$ and j's local clock reads $s_j(k)$ match with each other. That means the view of the TDMA cycle should be the same for all the nodes.

When the system is designed, each slot in the TDMA cycle is a priori assigned to a node for sending a message. Each node knows the structure of a TDMA round, i.e., which slot in the TDMA cycle is assigned to which node. Each node knows the duration of the slot assigned to any node in the TDMA cycle. For the sake of simplicity, we assume that the duration of the slot assigned to each node is the same. We denote the duration by τ. Each node also knows the propagation delay for the messages from any node in the system. During the startup process, each message contains the identifier of its sender.

In the startup process, a node can be in one of the following three states: *listen*, *coldstart* and *sync*. After the system is powered-on, a node in the system performs some internal initialization and moves to the *listen* state. Thus, different nodes enter the *listen* state at different point of time. There are two possible processes that a node can follow in the *listen* state: integration and coldstart. A node can follow the integration process only if some nodes are already in the *sync* state or in the *coldstart* state. In that case, the node can directly move from the *listen* state to the *sync* state by extracting necessary timing information from a message sent by a node that is already in one of those states. Otherwise, the node has to follow the coldstart process.

Before describing the coldstart process, let us explain how a node can extract its local state from a message obtained from another node. If node j gets a state message from node i at its local time t_1 and if the propagation delay for a message from node i is δ_{ij} then node j can conclude that the slot for node i in the TDMA round starts at $(t_1 - \delta_{ij})$ time according to its local clock. Then its own slot should start at time $t_1 - \delta_{ij} + \{(j-i)\ mod\ N\} \times \tau$ according to its local clock, where N is the number of nodes in the system and τ is the slot duration. We assume that each slot has an equal duration τ. If the slots are of different durations then also the node can easily calculate the starting time of its own slot by knowing the duration of the slots for other nodes. In the same way,

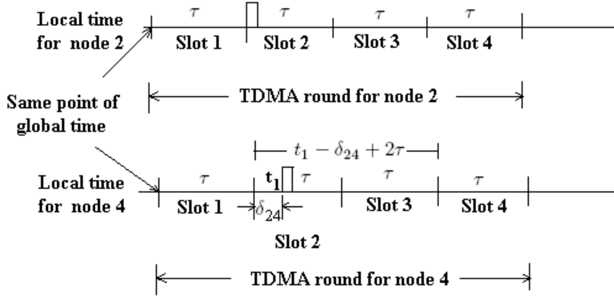


Fig. 3. TDMA round extraction by node 4 from the message of node 2

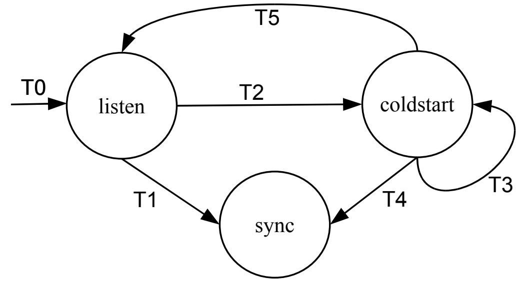


Fig. 4. State diagram of TTA startup algorithm

the node can calculate the starting time of the slots for other nodes too. Figure 3 shows how node 4 defines the TDMA round according to its local time by getting the state message from node 2.

In the startup algorithm, one of the nodes has to take the lead to initiate the process of timing synchronization by defining the global timing state based on its local timing. This process is called the coldstart process. The node which initiates the coldstart process is called the *leading coldstart node*. It goes to the *coldstart* state when its listen timeout occurs and it is unable to move to the *sync* state through the integration process. When a node moves to the *coldstart* state, it broadcasts a message that includes the suggested global timing state of the system as defined based on the local timing of the node. This message is called the *coldstart message*. In a fault-tolerant time-triggered system, a subset of the nodes are defined as *coldstart nodes*; these nodes are the ones allowed to participate in the coldstart process. All other nodes can only go to the *sync* state through the integration process. As there are more than one coldstart node, it is always possible that some of these nodes send their coldstart message almost at the same point of time. When two nodes send the coldstart messages simultaneously defining the system state as their own, a contention occurs and the coldstart process fails. Different time-triggered protocols use different strategies to resolve contention and select the one that should define the state of the system. We shall discuss them in detail when we describe the three time-triggered protocols individually in the following subsections.

A. The TTA Startup Algorithm

In a TTA cluster, two separate algorithms are used for the startup of nodes and guardians. They are similar in nature, though the startup algorithm for guardians is more complicated. Here we shall briefly describe the startup algorithm for the cluster nodes. For the detailed description of the algorithms, readers are referred to [17], where the authors present a fault-tolerant synchronous architecture, and an algorithm that ensures the transition from asynchronous to synchronous operation within a bounded duration at startup. They analyze the algorithm for different failure cases that violate the transition within a fixed duration, and provide guardian-based solutions for these failures.

In the TTA startup algorithm, all the nodes are coldstart nodes, i.e., all the nodes participate in the coldstart process. The state-machine of the startup algorithm executed in the nodes is shown in Figure 4. Each node i $(i = 1$ to $N)$ has two unique timeout parameters, τ_i^{listen} and $\tau_i^{coldstart}$ defined as follows:

$$\tau_i^{listen} = 2\tau^{round} + \tau_i^{startup}$$
$$\tau_i^{coldstart} = \tau^{round} + \tau_i^{startup}$$

where τ^{round} represents the duration of the TDMA round and $\tau_i^{startup}$ denotes the duration between the beginning of a TDMA cycle and the time when the slot for node i starts. If τ denotes the duration of each slot then

$$\tau^{round} = N\tau$$
$$\tau_i^{startup} = (i-1)\tau$$

After the system is powered-on, a node performs some internal initialization and moves to the *listen* state (T0). In this state, the node listens for a time period τ_i^{listen} to determine if there is already a set of nodes in the *sync* state. The nodes which are in the *sync* state periodically transmit an i-frame that carries the global TDMA cycle structure. If the node in the *listen* state receives such an i-frame, it adjusts its state to the frame contents and moves to the *sync* state (T1). If a node cannot move to the *sync* state by receiving an i-frame, there are two possibilities. In the beginning of the coldstart sequence, each node tries to listen to a coldstart message (cs-frame) from another node. A cs-frame contains the TDMA structure suggested by the sending node. When a node completes receiving a cs-frame, it enters the *coldstart* state and resets its local clock to δ_{cs} which denotes the transmission duration of the cs-frame. Thus, all nodes that received the cs-frame in their *listen* state, have the same view of the global time. Each node that receives neither an i-frame nor a cs-frame during the *listen* phase, enters the *coldstart* state at the expiry of its listen timeout (T2), resets its local clock to 0 and broadcasts a cs-frame. Thus after δ_{cs} units of time of the transmission of the cs-frame, the local clock of the sending node will be synchronized to the local clocks of the set of receiving nodes. Each node k in the *coldstart* state waits to receive another cs-frame or i-frame till its local clock reaches the value of its individual cold-start timeout $\tau_k^{coldstart}$. If it does not receive such a frame by that time, it resets its local clock and again broadcasts a cs-frame (T3). If the node receives a frame with

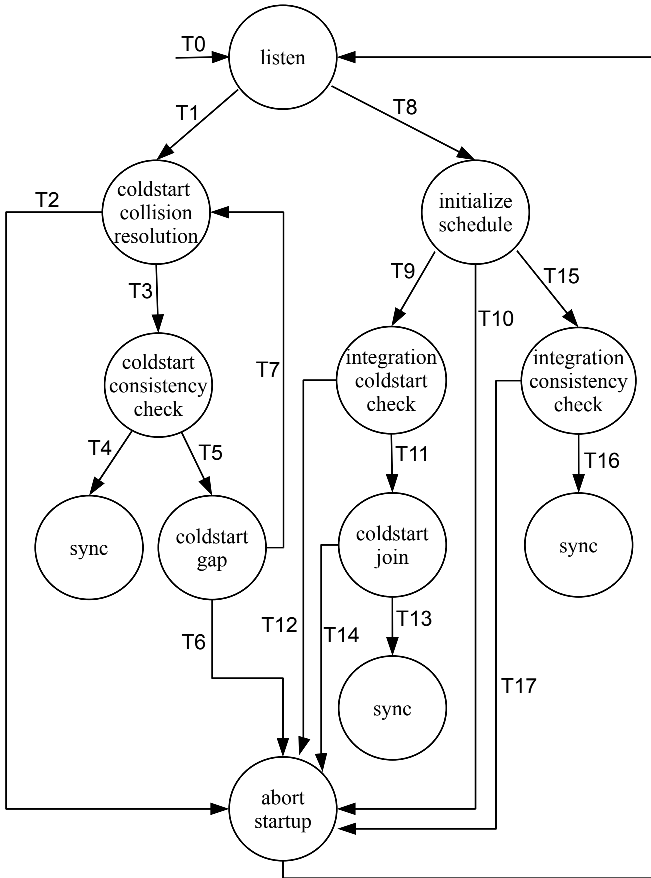


Fig. 5. State diagram of FlexRay startup algorithm

which its own view of the TDMA structure matches, it enters the *sync* state (T4). Otherwise, if the node receives a faulty frame, or detects noise in the channel, or receives a valid frame whose TDMA structure does not match with the node's view of TDMA structure, the node goes back to the *listen* state (T5).

B. The FlexRay Startup Algorithm

In this section, we briefly describe the startup algorithm for FlexRay, skipping some of the details of the algorithm. For a detailed description of the algorithm, readers are referred to [14]. A state-machine diagram for each node in the startup algorithm of FlexRay is shown in Figure 5. Here a subset of nodes acts as coldstart nodes. Other nodes can only integrate after a sufficient number of coldstart nodes are synchronized. Once the system powers on, a wake-up algorithm is executed to make all the nodes ready to participate in the startup algorithm (T0). At the beginning of the startup algorithm, all the nodes are in the *listen* state. There are three paths a node can follow from the *listen* state: that of the leading coldstart node, that of coldstart nodes that cannot be leading coldstart nodes, and that of the non-coldstart nodes.

In the *listen* state, a coldstart node listens to the attached channel and tries to receive a valid startup frame for at least two TDMA rounds. If no communication is received during this time, the node transits to the *coldstart collision resolution* state (T1) and commences the coldstart attempt by transmitting a Collision Avoidance Symbol (CAS). Since each coldstart node can perform a coldstart attempt independently, it may happen that several nodes simultaneously come to the *coldstart collision resolution* state and transmit the CAS symbol almost at the same time. In this case a contention takes place, which is resolved during the first four cycles after CAS transmission. As soon as a node that initiates a coldstart attempt receives a CAS symbol or a frame header during these four cycles, it re-enters the *listen* state (T2). Consequently, only one node remains in the *coldstart collision resolution* state and acts as the leading coldstart node. After spending four rounds in *coldstart collision resolution*, the leading coldstart node performs a *coldstart consistency check* (T3) and sees whether there exists at least one node that has used its messages for integration. If that is not the case, it tries to keep up communication for one more TDMA round until it finally concludes that synchronized operation has been reached (T4). If the leading coldstart node does not get any response, it enters a *coldstart gap* (T5) where it waits for another TDMA round. In the meantime, if a CAS or a correct header is received, the node goes back to the *listen* state (T6), otherwise it re-enters *coldstart collision resolution* at the end of the round (T7) and tries to synchronize with other nodes again.

In the *listen* state, if a coldstart node receives a valid startup frame, it immediately goes to the *initialize schedule* state (T8), and waits for the second startup frame to derive the TDMA schedule from the two received frames. When the second valid frame is received, the node moves to the *integration coldstart check* state (T9); otherwise, it goes back to the *listen* state (T10). In the *integration coldstart check* state, the node waits for two more startup frames to check its own clock correction mechanism. If the clock correction becomes successful with these new frames, the node jumps to the *coldstart join* state (T11), otherwise it goes back to the *listen* state (T12). In the *coldstart join* state, the node starts sending its own startup frames. If the node still receives valid startup frames from other nodes, it goes to the *sync* state (T13), otherwise it goes back to the *listen* state (T14).

A non-coldstart node in the *listen* state listens to its attached channels and tries to receive a valid start frame. It immediately goes to the *initialize schedule* state on receiving a valid startup frame (T8), and waits for the second startup frame to derive the TDMA schedule from the two received frames. When the second valid frame is received, the node goes to the *integration consistency check* state (T15), otherwise it goes back to the *listen* state (T10). In the *integration coldstart check* state, the node waits for two more startup frames to perform its own clock correction mechanism. If the clock correction is successful with these new frames, the node jumps to the *sync* (T16) state; otherwise, it goes back to the *listen* state (T17).

C. The TTCAN Startup Algorithm

In a TTCAN system there are more than one coldstart nodes; these nodes are called *potential time masters*. The number of potential time masters is predefined and a strict ordering on identifiers of the potential time masters is assumed. Based on their identifiers, their priorities are determined. We have

developed a model of the startup algorithm of TTCAN from its textual description available in [18]. The state diagram given in Figure 6 depicts the behavior of the potential time masters (for more details refer to [9]). All the potential time masters start executing the startup algorithm in the *listen* state. In this state, a potential time master listens to the bus until its listen timeout to check whether a reference message has already been sent. If no reference message has been sent previously, it sends a reference message and becomes the current active time master (T1). If more than one potential time master attempt to transmit reference messages at the same time, a contention occurs, and it is resolved by the arbitration process of CAN. This ensures that one potential time master becomes the active time master, and this time master defines the TDMA schedule. The objective of the TTCAN startup algorithm is that the non-faulty potential time master with the highest priority should eventually become the active time master. Note that the potential time master which becomes the active time master in the beginning, may not necessarily have the highest priority. Potential time masters which find a reference message already in the bus in their *listen* state, jump to the *wait* state (T2) and wait for the next reference message. When a reference message is received, a potential time master in the *wait* state becomes synchronized with the current active time master by extracting the synchronization information from the reference message. It also compares its own priority with that of the current active time master (the priority of the current active time master can be obtained from the reference message). If the priority of the potential time master is less than the priority of the current active time master then the potential time master goes to the *sync* state (T3), otherwise it comes to the *arb_in_next_basic_cycle* state (T4). At the beginning of the next basic cycle, the potential time master which is in the *arb_in_next_basic_cycle* state and has the highest priority among the current active time master and all the potential time masters in the *arb_in_next_basic_cycle* state, replaces the current active time master and becomes the active time master (T5). The previous active time master and all the potential time masters in the *arb_in_next_basic_cycle* state besides the new active time master go to the *sync* state (T6, T7). The new active time master maintains the same TDMA structure as defined by the previous active time master. The startup process comes to an end eventually when the highest priority potential time master becomes the active time master.

A state diagram of the controllers which are not potential time masters is shown in Figure 7. These are called general controllers. A general controller listens to the bus in its *listen* state. When the reference message is received, it synchronizes with the current active time master by extracting the synchronization information from the reference message, and moves to the *sync* state (T1).

IV. Formal Modeling of Faults

Most safety-critical systems which employ time-triggered architectures often demand a high degree of reliability as they are expected to operate in the presence of faults. Modern aircraft systems generally have to operate for several days

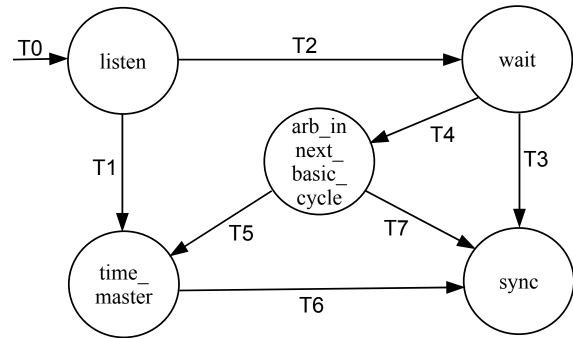


Fig. 6. State diagram of a potential time master for fault-free TTCAN startup

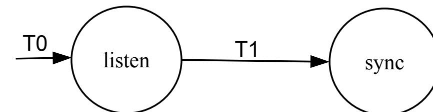


Fig. 7. State diagram for a general controller in startup algorithm of TTCAN

with some components being faulty [19]. Redundancy and replication are common means of achieving fault-tolerance. Redundancy consists of the replication of the entire bus, of the interconnect and/or the interfaces, or decomposition of these elements into smaller subcomponents which can be then replicated.

The source of a fault in most of the cases is a hardware malfunction. For example, a weak power supply may send intermediate voltages that can be treated neither to be 0 nor to be 1. In defining the fault and modeling the behavior of the system in the presence of the fault, a verification engineer has to rely upon the knowledge of the system engineer about different erroneous behaviors that the underlying hardware can induce in the system due to a known set of hardware malfunctions. While the experience of the engineers greatly help defining the kinds of faults that may arise during the lifetime of the system, it can never be guaranteed that all possible faults have been considered during the design of the fault-tolerant system. Thus the fault-tolerance behavior of the system is defined based on some well-defined fault-hypothesis. The goal of the formal verification is to ensure that if the system satisfies the fault hypothesis, the system does exhibit the desired behavior. In the following subsections, we discuss fault hypothesis and fault-modeling for time-triggered architectures.

A. Fault Hypothesis for Different Time-Triggered Architectures

Any design of a fault-tolerant system must be evaluated against an explicit fault hypothesis which specifies the number and type of faults it is intended to tolerate. Any fault-tolerant architecture will fail if subjected to too many faults. In such a case, it is assumed that there are components in the architectures that can be independently afflicted by faults. These components are called fault containment units (FCUs) [20]. The fault hypothesis should be able to recognize these different FCUs individually in the design. The division

of an architecture into separate FCUs must ensure that there would not be any propagation of faults from one FCU to another, and there must be no common mode of failures where a single physical event produces faults in multiple FCUs. While designing a fault-tolerant system, each node and each communication channel is considered as one FCU. A faulty node may be of two types: *fail silent* and *Byzantine faulty*. If a faulty node is fail silent, then the node does not participate further in any communication in the network. However, if a node is Byzantine faulty, the node can still send an arbitrary set of messages at any arbitrary time for an arbitrary duration. A faulty channel can send noise to a subset of nodes at any time, though it cannot create any correct message. Thus a faulty channel shows Byzantine behavior with certain limitations.

The three different architectures that we consider make use of different fault hypotheses with respect to startup algorithms.

1) TTA: The fault hypothesis of the basic bus-based TTP/C protocol is discussed in [21]. A single fault hypothesis is adopted by TTA. Fault injection studies in [22] show that adding central guardians to the architecture [23] can be effectively used to achieve required fault tolerance in the aerospace and automotive industries. It is claimed that TTA can tolerate one faulty component if central guardians are used. The faulty component may either be a faulty node or a faulty channel (including its guardian). A faulty node can be fail silent or can show Byzantine behavior. It can send an arbitrary signal with arbitrary frequency for an arbitrary amount of time. A faulty channel, unlike a faulty node, cannot create correct frames. The faulty channel does not delay frames for an arbitrary duration. A faulty channel can deliver noise instead of the correct signal to a destination. For a more elaborate discussion on the fault hypothesis of TTA, the reader is referred to [6].

2) FlexRay: In FlexRay, a single-fault hypothesis is considered: at most one node or one channel can be faulty. The faulty node may become the leading coldstart node, subsequently restarting the algorithm after sending a few coldstart messages. Hence, a faulty node in the FlexRay system can behave awkwardly, and exhibit Byzantine behavior. When a channel is faulty, it can produce a Collision Avoidance Symbol (CAS). CAS symbols are supposed to be sent only by coldstart nodes at specific time points during the protocol run. However, as the CAS is a simple low-pulse signal of a specified duration, the faulty channel itself can produce the signal, thus disrupting a protocol run.

3) TTCAN: In TTCAN, all the potential time masters take part in startup, and eventually one of them with the highest priority becomes the active time master. The responsibility of the active time master is to send a reference message at the beginning of each basic cycle. If the active time master is no longer capable of sending the reference message, then it is assumed to be faulty. A faulty time master can only become fail silent, Byzantine behavior from a faulty time master is not expected. For the proper operation of the startup algorithm, at least one time master needs to be alive. Thus the fault hypothesis for TTCAN startup algorithm is that at most $N-1$ time masters can be faulty during startup, where N is the total number of time masters in the system.

Note that any node other than the active time master may be faulty at any time during operation. But that is of no relevance to the functionality of the startup algorithm.

B. Fault Modeling

During fault modeling, it is assumed that the types of faults which are expected to occur during operation are known a-priori. A fault tolerant mechanism is activated by the occurrence of a fault. As faults occur rarely, testing and debugging a fault-tolerant mechanism is cumbersome. Formal verification comes to our aid by enabling us to verify the behavior of a system during the execution of an algorithm in the presence of various known faults. The success of providing a correctness assurance of a system in the presence of faults largely depends on the way the faulty behaviors are modeled. The procedure of introducing faulty behaviors in the model of a system is called *fault injection*. In a fault-injected model, the behaviors are affected not only by the inputs to the models, but also by various injected faults.

For fault-tolerant systems the nodes are modeled as processes. Each process consists of a finite set of actions that capture the transitions of the systems from one state to another under different circumstances. Each action consists of a *guard* and a *transition* statement, where a guard is a boolean expression over the current state variables, and a transition statement is a set of assignments by which the state variables of the process are updated to the next state values. An action is enabled at a state if and only if, its guard evaluates to true in the current state.

Ideally, the fault model should be exhaustive to capture all kinds of possible faults, and the model checker should be able to inject them in all possible ways. Yokogawa et. al. [24] presents a methodology to describe all possible faults in a system through the use of guarded commands. Various types of faults can be described by a set of actions, designated as *fault actions* in [24]. The transition statement of a fault action uses a boolean variable $faulty$ for announcing the fault, which is initially set to false. When the fault occurs, the boolean variable is set to true.

Let us now explain how a crash failure can be modeled. Let us denote the guard for a particular transition as "$guard$". We use the following notation to denote that a particular transition can be enabled when $guard$ is true.

$$\langle guard \rangle \Rightarrow \langle transition \rangle$$

It represents a behavior of the process when it is not faulty. If we want to model crash failure for this process, a boolean variable *faulty*, initially set to false is added to the guard. Now the guarded transition will look like:

$$\neg faulty \wedge \langle guard \rangle \Rightarrow \langle transition \rangle$$

When the boolean variable *faulty* is false, the process can enable this particular transition. However, when it is true, the transition cannot be enabled. This captures the desired behavior of a process which may crash.

To model Byzantine failure [25], we need to consider guarded transitions modeling the Byzantine behaviors of the

process. Let $bguard$ represent the guard for the transition of a process which may show Byzantine behavior:

$$\langle bguard \rangle \Rightarrow \langle transition \rangle$$

Unlike crash failure, the Byzantine behavior of a process is enabled when the failure occurs. The guarded transition in this case will be

$$faulty \wedge \langle bguard \rangle \Rightarrow \langle transition \rangle$$

The boolean variable $faulty$ becomes true when the process starts showing a Byzantine behavior. So, when the process is not Byzantine faulty (i.e., the boolean variable *faulty* is false), this behavior is not present.

Finally, the following fault action is added to the set of guarded transitions of the process.

$$\langle fault \rangle \Rightarrow \langle faulty = \textsf{true} \rangle$$

The guard of this transition $fault$ specifies when the fault will be triggered. A simple true in the guard will cause the failure of the process at an arbitrary time when this transition is selected non-deterministically.

If a process is involved both in a crash failure and a Byzantine failure, then two separate flags need to be used. Let us denote the flags for crash failure and Byzantine failure for a process as $cfaulty$ and $bfaulty$, respectively. The following two guarded transitions are used to represent the fault actions:

$$\langle cfault \rangle \Rightarrow \langle cfaulty = \textsf{true} \rangle$$

$$\langle bfault \rangle \Rightarrow \langle bfaulty = \textsf{true} \rangle$$

where $cfault$ and $bfault$ represent the condition under which the crash failure and Byzantine failure for the process occur respectively. Now a normal operation of the process is represented by the following guarded command:

$$\neg cfaulty \wedge \neg bfaulty \wedge \langle guard_1 \rangle \Rightarrow \langle normal_transition \rangle$$

The Byzantine behavior of the process is represented by the following guarded command:

$$\neg cfaulty \wedge bfaulty \wedge \langle guard_2 \rangle \Rightarrow \langle byzantine_transition \rangle$$

Thus, only normal behavior of the process is possible when both the flags are false. In the case of crash failure (i.e., $cfaulty$ is true) no transition of the process is possible. In the case of Byzantine failure (i.e., $bfaulty$ is true), the process can show only Byzantine behaviors, unless a crash failure has occurred.

V. Formal Models of time

Startup algorithms are real-time protocols. Hence, before discussing the formal verification of the correctness of these algorithms let us describe different ways of modeling time for real-time systems. The most commonly used model is the well-known *timed automata* model introduced by Alur and Dill [26]. In this model, the state variables of the system include a set of clock variables, and a set of discrete variables. Initially, all the clock variables are set to zero. The transition relation consists of two types of transitions - *time progress* transition and *discrete* transition. In time progress transition, all the clock variables are increased by an equal amount, but the values of the discrete variables do not change. During a discrete transition, the clock variables do not change their values; the values of a subset of discrete variables are updated appropriately. However, the ordinary clock-based models have several drawbacks. As the clocks vary continuously with time and the time value with which the clocks are updated during a time progress transition can be arbitrarily small, ensuring progress becomes difficult. The transition system has infinite trajectories in which no discrete transition ever occurs and these undesirable trajectories cannot be easily excluded. As a result, time remains bounded in some trajectories and analyzing liveness properties become difficult. This led to the consideration of models used in discrete event systems, like timeout-based models, calendar-based models and so on. These new models have been successfully used in a number of experiments dealing with the verification of startup algorithms of time-triggered architectures.

A. Timeout-based Models

In a *Timeout-based model* [27], continuous dynamics is avoided by maintaining a timeout variable for each process that keeps track of the time when the next discrete transition for that process occurs. Let us denote the set of timeout variables by T. The state variables of these systems also include a variable t, the time variable that keeps track of the current time. All these variables are real-valued.

In the initial state, all the timeouts are set to values higher or equal to the value of the current time. When the value of the time variable t is less than all the timeout variables, *time_progress* transition is triggered, which increases the value of the time variable to the minimum value of all the timeouts. This transition leaves all the other state variables unchanged. A *discrete* transition occurs when the value of a timeout is equal to the value of the current time. During this transition, the timeout variable with the least value is updated to an appropriate time value in the future. The value of the time variable is left unaltered. A subset of other state variables are updated appropriately during a *discrete* transition. Note that the value of t is always less than or equal to that of timeout variables.

The timeout-based modeling approach is appropriate for modeling systems where the processes communicate via shared variables. It is not applicable in situations where a message takes a certain amount of time to reach the receiver from the sender. In a timeout-based model, process p_i can determine on its local timeout, that is, it can decide when to take its next transition. Other processes have no access to p_i's timeout. However, the variable denoting the message delay cannot be associated with a particular process, and a possible way of modeling a message is to represent it by a global variable. Calendar-based modeling is apt for this purpose.

B. Calendar-based Models

Asynchronous communication with bounded message passing delay can be very easily modeled by using *Calendar* as a

global data structure [5]. When a message is transmitted by a process, it is added to the calendar as an event e_i scheduled to occur at time t_i, (t_i denotes the time when the message will be delivered to the receiver of the message). When the receiver receives the message, the event is removed from the calendar. In short, a calendar can be seen as a set of messages that have been sent, but are yet to be received. Each message is also labeled by the time when it is received by the receiver.

The state variables of a calendar-based system include a variable indicating the current time, a set of variables denoting the timeouts for individual processes, and a calendar. In the initial state, the values of the timeout variables and the time corresponding to any event in the calendar are more than the current time. When the current time is less than all the timeouts and the time entries corresponding to the events present in the calendar, a *time progress* transition is triggered. During this transition, the time variable is increased to the minimum of all the timeouts and the time corresponding to the events in the calendar. A *discrete* transition occurs, when the current time is equal to the minimum of timeouts or the minimum of all the time values in the calendar, whichever is less. During a *discrete* transition, the time variable does not change. If the time variable is equal to the minimum of all the timeouts, but less than any time entry in the calendar, then calendar entries do not change and the minimum timeout is updated to an appropriate future value. If the current time is equal to the minimum of the time entries in the calendar, but less than the minimum timeout, then the calendar entry with the minimum time value is removed from the calendar, and a corresponding message is delivered to the appropriate receiver. If the current time is equal to the minimum timeout and the minimum of the time entries in the calendar at the same time, then either the event corresponding to the timeout or the calendar is selected arbitrarily.

C. Synchronous Calendars

While calendar-based modeling is appropriate to model systems where communication is asynchronous with bounded message propagation delay, it cannot be directly applied to cases of synchronous communication. On the other hand, the timeout-based model is not adequate in situations where any discrete transition occurs in the event of receiving some message. To capture synchronous communication, Saha et al. [9] introduced a data structure called *Synchronous Calendar*. A synchronous calendar contains information about the sender of a message and a flag for every process present in the system. Formally, for an n-process system a synchronous calendar is a tuple $SC = \langle e, (v_1, \cdots, v_n) \rangle$, where $v_i \in \{0, 1\}, 1 \leq i \leq n$. Note that $v_i = 1$ denotes the fact that on the occurrence of event e, the message is delivered to process i. Once a synchronous message is delivered, the calendar becomes empty (there is no entry in the calendar). An empty calendar is denoted as EMPTY.

A system in which all the communications are synchronous in nature can be modeled by using a set of timeouts for each of the processes along with a synchronous calendar. In the initial state, the value of time is less than all the timeouts and the calendar is EMPTY. A *time progress* transition is enabled when the synchronous calendar is EMPTY, and time is less than each of the timeouts. A *discrete* transition may occur due to timeout or due to a synchronous communication. If the *discrete* transition is due to timeout, then the minimum-valued timeout corresponding to the particular process is increased to an appropriate value in future at which the next timeout for the same process will occur . During this transition, a synchronous message may be added to the synchronous calendar (the calender becomes non-EMPTY). If the transition is due to a synchronous communication, the message is delivered to the targeted recipients and the synchronous calendar becomes EMPTY.

D. Clockless Modeling of Timeout and Calendar Based Models

Although the timeout and calendar-based models can be used to capture dense time semantics efficiently without using a continuously varying clock, these models are difficult to use for finite state model checking as the valuations for the global clock t and the timeout variables in $\mathcal{T}$ are not bounded by a finite domain. Saha et al. [7] proposed a finitery reduction technique, which effectively reduces the timeout and calendar-based transition systems with discrete dynamics into finite state systems, which in turn, can be verified using finite state model checkers. This model does not require any explicit clock, and thus is called a *Clockless Model*. When no discrete transition is possible in the system due to the fact that the discrete transitions for all the processes are scheduled in the future, all the timeouts are decreased by the minimum of the timeouts. In this way, at least one of the timeouts becomes zero. A process is allowed to take a discrete transition when its timeout becomes zero; at this point the process updates its timeout to an appropriate value larger than 0 and does other necessary jobs. This approach can be easily extended to the calendar-based models as well.

If the timeouts are always incremented by finite values then it is guaranteed that the value of a timeout will always be in a finite domain. The finitery reduction is effective only under discrete dynamics since with dense modeling such a reduction, though it partitions an infinite state-space into many finite regions, would still contain infinitely many points resulting in infinite permissible paths.

There are cases when a timeout increment cannot be bounded by a finite value. For example, a process may have to wait for an external signal before its next discrete transition occurs. In this case, this discrete transition of the process does not depend on its own timeout. The timeout of the process is set to a relatively large value, so that it does not affect the next discrete transitions of other processes. In another situation, it may be required that the next discrete transition of a process may happen at any time in the future; for example, the process may be in a sleeping mode and can wake up at any future point of time. In this case all that is needed is to limit the value of the timeout without omitting any of the possible interleaving of the process steps. To do that we limit the timeout value in $[0, M+1]$, where M is the maximum of all the integer

constants that are used to define the upper limit of different timeouts for different processes in the system.

The verification technique proposed above avoids the complexity of induction-based methodology for infinite state (bounded) model checking. Moreover, this technique not only enables one to verify safety invariants for real-time systems, but also helps one verify liveness properties for real-time systems, which is not possible by using induction with infinite state model checkers.

E. Suitability of Different Time models to Startup algorithms

One question that naturally arises is the following: which type of time model is suitable for startup algorithms? Any startup algorithm involves message passing between the nodes, and the nature of message may be asynchronous or synchronous. The time model that can capture the time of message propagation is calendar-based, while the synchronous calendar-based model can capture communication using synchronous messages. Although the timeout-based model is efficient enough to model timed systems involving communication by shared variables, it is difficult to capture message propagation time in this model. On the other hand, in [6] it is reported that an ordinary clock-based model of the TTA startup algorithm has billions or even trillions of reachable states, though the model captures only discrete time. This indicates how complex the verification of a clock-based model would be under the assumption of continuously varying time. Hence, the standard or synchronous calendar-based model, depending on the nature of the message passing, appears to be the obvious choice for startup algorithms.

VI. Correctness Properties

To ensure the correctness of a startup algorithm, the following three properties are of primary interest: *safety*, *liveness* and *timeliness* [6]. As a startup algorithm is a distributed one, verifying *safety* and *liveness* properties are essential to establish its validity. Verification of the *timeliness* property ensures that the startup process is completed within a bounded time. The properties are formally captured using Linear Temporal Logic (LTL) [28]. In LTL, the temporal operator $\Diamond$ stands for (classical) "eventual" modality and and the temporal operator $\Box$ stands for (classical) "always" modality, where $\Box\phi \stackrel{\text{def}}{=} \neg\Diamond\neg\phi$ for any formula ϕ, where $\neg$ is the logical operator not. For a logical formula ϕ, $\Diamond\phi$ intutively means that the formula ϕ becomes true sometimes in the future, and $\Box\phi$ means that the formula is true all the time during the system's operation. A property of the form of $\Box\phi$ is often called an *invariant* property. Below we describe the three properties in the context of startup algorithms.

Safety. Whenever any two nodes are in their *sync* state, the nodes agree on the slot time.

Recall that the local time of node i when the slot for node k begins is denoted by $s_i(k)$. Also, the state of a node i at a particular time instant is represented by $state_i$. Thus we represent the safety property as follows:

$$\Box\Big(\forall i, j.((state_i = \mathit{sync}) \wedge (state_j = \mathit{sync})) \Rightarrow (s_i(k) = s_j(k))\Big),$$

where i and j vary over a finite set of nodes.

Liveness. All non-faulty nodes will eventually reach the *sync* state.

When a node finishes its execution of the startup algorithm successfully, it is said to be in the *sync* state and if a node is found faulty, it is said to be in the *faulty* state. Then the liveness property can be represented as:

$$\Diamond\Box(\forall i.(state_i = \mathit{sync} \vee state_i = faulty))$$

Timeliness. All non-faulty nodes will reach the *sync* state within a bounded time.

We define the startup time to be the time required for all the non-faulty nodes to come to the *sync* state after the system is powered on. Let us denote it by $startup_time$. This is captured by a global variable in the model, which depends on system parameters. Further, the time when the startup algorithm is expected to finish as mentioned in the specification, is denoted by $specified_startup_time$. Now the timeliness property can be written as follows:

$$\Box(startup_time \leq specified_startup_time)$$

The *Timeliness* property can be considered as a bounded liveness property. If the required time for the startup process to be completed is mentioned in the specification of the startup algorithm, it is enough to verify the *timeliness* property, and the verification of the liveness property is not required. But for some startup algorithms (for example, TTCAN), the specification does not mention the maximum time required for the completion of the startup process, which necessitates the verification of the liveness property.

VII. Verification Approaches

Verification of startup algorithms have so far relied on the model checking techniques [29]. Though researchers have used theorem prover PVS [30] to verify other algorithms for time-triggered architectures [31], [32], [33], model checking is the only technique that has been applied to the verification of startup algorithms. Below we discuss verification techniques used in a few model checkers, e.g., SPIN [34], SAL [35], [36] and UPPAAL [37], which have been used to verify the startup algorithms.

A. Verification through Finite State Model Checking

The model checking approach was initially introduced for analyzing finite state programs [38], [39], [40], [41], [42]. Finite state programs are the ones in which the variables range over finite domains. A finite state program can be modeled as a finite propositional *Kripke structure* [29], which can be specified by a formula in propositional temporal logic. The problem of verifying whether a program is correct reduces to checking if the program, viewed as a Kripke structure, satisfies the propositional temporal logic specification [43], which is more popularly known as verification by *model checking*. One of the merits of model checking is that the verification can be performed algorithmically by performing exhaustive search on the graph induced by the finite state-space of the Kripke structure. SPIN [34] is an explicit state model checker that

searches for all the state spaces exhaustively. SPIN employs some optimization techniques which either reduce the number of reachable system states that have to be searched to verify properties, or reduce the amount of memory needed to store each state. Its *partial order reduction* technique [44] falls into the first category, while the *bitstate hashing* technique [45] is in the second category.

B. Verification through Infinite State Model Checking

Model checking has been quite successful for verifying finite state systems. However, using finite symbolic representation of infinite state variables, infinite state systems can also be verified using model checking techniques. The bounded model checking technique [46] has been developed for such purposes. Bounded model checking makes use of powerful Boolean satisfiability (SAT) solvers for the verification of LTL [29] properties. In this technique, the states are encoded as Boolean variables and a program is unrolled a finite number of steps for some bound k, and an LTL property is checked for counterexamples of length k. That is, given a program $\mathcal{P}$ over an infinite state, a linear temporal logic formula ϕ with domain specific constraints over program states, and an upper bound k, this procedure determines if there is a falsifying path of length k to the hypothesis that $\mathcal{P}$ satisfies the specification ϕ. This model checking problem can be shown to be equivalent to the satisfiability of Boolean constraint formulas. The latter is usually the technique applied for verifying properties using infinite state model checkers like SAL. Since the modeling of startup algorithms calls for capturing dense time, an infinite state model checker is useful for verifying them. SAL provides the option of using bounded model checking in the context of a startup algorithm, modeled as an infinite state system.

C. Model Checking of Timed Automata

In a timed automata, the clocks are real-valued. Thus, the state-space of a timed automata is infinite. The verification of timed automata is based on the construction of a finite state graph called *region graph* using a notion of region equivalence of states over clock assignments [26]. A more efficient representation of the state-space for timed automata is called *zone-graph* [47], which provides a more compact representation of the state-space. Uppaal [37] is a toolbox for the verification of real-time systems modeled as networks of timed automata.

D. Suitability of Different Verification Approaches to Startup algorithms

Calendar or synchronous calendar based continuous time models can be used to verify only invariant (safety and timeliness) properties by using the induction based infinite-state model checking method in SAL. The difficulty of using the induction based method in SAL is that the proofs do not scale well. An invariant property often cannot be proved at induction depth 1. Sometimes the properties are not inductive and need the support of auxiliary lemmas. In [5], a safety property for the TTA startup algorithm with only 2 nodes has been proved by using 3 additional lemmas. Though the verification diagram-based abstraction method proposed in [48] has been used to prove the safety (invariant) property for scaled up models (upto 10 nodes), liveness properties still remain beyond the scope of this approach. If we restrict our models to a discrete time domain and use the clockless modeling technique as described in [7], the verification can be performed by using a finite state model checker like SPIN which is more user friendly than SAL, and relieves the user from the burden of finding additional lemmas, and abstraction based on a disjunctive invariant [48]. In terms of scalability, the discrete time finite state verification of TTA in SPIN [7] is almost comparable to the verification of TTA based on the verification diagram-oriented abstraction method [5]. Moreover, liveness properties can be verified in the finite state verification processes.

VIII. Verification of Startup Algorithms

In this section, we shall discuss modeling and verification of the startup algorithms carried out by different researchers in the past. We have identified four characteristics for analyzing these algorithms, viz., *time modeling*, *fault modeling*, *correctness properties* and *performance issues*. For each verification experiment, we would describe how time had been modeled, what the fault hypothesis was, and what properties had been verified. Moreover, we discuss the number of system components (number of nodes and number of channels) that have been considered as parameters in the verification of different algorithms. The verification efforts of all these startup algorithms are summarized in Table I.

A. TTA

There are three pieces of work on the model checking of the TTA startup algorithm in the literature. The first one is by Steiner *et. al.* [6] using SAL model checker in the framework of a discrete time model. The model has a counter which can take only integer values to capture the propagation of time. It is similar to a clock-based model, where clocks take only integer values. The model does not take into account the message propagation time. Each node is associated with one such clock, and a discrete transition can reset the value of a clock. In [6] it is reported that the model has billions or even trillions of reachable states, though it captures only discrete time. Exhaustive fault simulation has been undertaken in this work. This model of the TTA startup algorithm ensures a safe and timely system startup in the presence of one faulty component, which can be either a faulty node or a faulty hub. They classify the possible outputs of such a faulty node into six fault degrees. A higher fault degree can also exhibit the behaviors of a lower fault degree. A fault degree of 1 allows a faulty node only to be fail silent, a fault degree of 2 allows a node to be fail silent or to send a correct cs-frame, a fault degree of 3 allows a node to be fail silent, or to send a correct cs-frame and an i-frame, a fault degree of 4 allows all the behaviors with fault degree 3 and a provision to send noise in the channel, a fault degree of 5 allows all the behaviors with fault degree 4, and a provision to send an incorrect cs-frame,

Ch A \ Ch B	quiet	cs_frame (good)	i_frame (good)	noise	cs_frame (bad)	i_frame (bad)
quiet	1	2	3	4	5	6
cs_frame (good)	2	2	3	4	5	6
i_frame (good)	3	3	3	4	5	6
noise	4	4	4	4	5	6
cs_frame (bad)	5	5	5	5	5	6
i_frame (bad)	6	6	6	6	6	6

Fig. 8. Fault Degree in fault simulation for TTA startup (borrowed from [6])

while a fault degree of 6 allows a node to be fail silent or send an arbitrary combination of correct or incorrect cs-frames and i-frames or noise. A faulty node may show different behavior in the redundant channels. To describe the possible behaviors of a faulty node in two redundant channels, a 6×6 matrix is defined in [6]. This matrix is called a "fault-degree" matrix and is reproduced in Figure 8. All the three properties mentioned in section VI have been verified in this work. Models with 3, 4, 5 and 6 nodes, and two channels in all the cases, have been examined.

Another work on the verification of the TTA startup algorithms is due to Dutertre and Sorea [5]. In this work, continuous time dynamics has been captured in SAL model checker by using Calendar Automata, which does take the propagation delay of message transmission into account. They have modeled the protocol with an active hub that is assumed to be reliable, but a single node may be Byzantine faulty, and may attempt to broadcast arbitrary frames at any time. The only property that has been verified in this work is the safety property. By using induction, the authors verify the safety property for only two nodes by strengthening the safety invariant with three additional lemmas, but induction fails for more than two nodes. For a higher number of nodes, they use a method based on a disjunctive invariant, originally proposed by Rushby [48]. This method constructs an abstraction of the transition system (or verification diagram [49]) based on a number of state predicates. By using this method, they have been able to verify the safety property for a TTA cluster of 10 nodes.

Verification of the safety and the liveness property for the TTA startup algorithm has been carried out using SPIN model checker in [7]. A clockless modeling technique has been adopted in this work in order to make the model a finite state one. This work considers only the fault-free aspect of the TTA startup algorithm. Verification has been carried out with the options of *exhaustive verification*, *bitstate hashing* and *partial order reduction* techniques offered by SPIN. By an exhaustive verification technique, the safety property can be verified for the TTA models with upto 5 nodes and the liveness property can be verified upto 4 nodes. Bitstate hashing helps verify both the properties for models with upto 9 nodes. For 10 nodes, the verification does not terminate even in 4 hours.

B. FlexRay

Model-checking of the FlexRay startup algorithm has been carried out by Steiner [8] using the SAL model checker. In this work, a discrete time modeling of the FlexRay startup algorithm has been undertaken, the time model being similar to the one in [6]. A single fault hypothesis is considered, i.e., either a node may be faulty, or a channel may be faulty. The node failure occurs when one of the nodes always becomes the leading coldstart node, and restarts the algorithm after sending some coldstart messages. Channel failure occurs when a channel produces a CAS symbol that causes all nodes to stay in the LISTEN state. As a remedy to the first problem, a simple guardian that traces the messages sent during the execution of the algorithm has been modeled. If a leading coldstart node fails to send the required sequence of coldstart messages, it is excluded by the guardian from participating further in the startup process. The second problem has been solved by introducing unique timeouts for the CAS GAP duration. All the properties mentioned in section VI have been verified. Only 4 nodes are considered for these experiments.

C. TTCAN

The modeling and verification of the TTCAN startup algorithm has been carried out in [9]. The SAL model checker was used to model and verify this startup algorithm. As signal propagation time is considered to be negligible in TTCAN [50], a Synchronous Calendar has been used for modeling time.

Both fault-free and fault-tolerant aspects of the startup algorithm of TTCAN have been captured in two different models. In the case of fault modeling, only crash failure for the active time master has been considered, as the crash failure in the other potential time masters and other controllers does not influence the startup mechanism. But for exhaustive fault simulation, the model has been developed in such a way that faults may arise in the active time master in all possible ways.

In this work, model checking has been carried out to verify both safety and liveness properties. The safety property has been proved by induction for models with up to 3 potential time masters for both fault-free and fault-tolerant startup with the help of one auxiliary lemma. Verification of the safety property has been successful for higher number nodes by using abstraction-based methodology, similar to what presented in [5]. For the fault-free model, verification of the safety property has been successful for models with up to 10 nodes, and for fault-tolerant startup, it could handle up to 9 nodes.

The infinite bounded model checker of SAL (sal-inf-bmc) does not support proof by induction for liveness properties, but supports bounded model checking. By using sal-inf-bmc, all the liveness properties have been verified up to depth 40 considering up to 10 potential time masters.

D. Other Protocols

In this survey we have concentrated mainly on TTA, FlexRay and TTCAN as they are widely adopted in the automotive and aerospace industries. However, there are other two time-triggered architectures for which the verification of

startup algorithms was undertaken, *viz.*, DACAPO [51] and SPIDER [52]. DACAPO (Dependable Architecture for Control of Applications with Periodic Operation) is an architecture for physically small safety-critical distributed systems. The communication in DACAPO is fully time-triggered, and bears a close resemblance to that of TTP/C. Lönn and Pettersson modeled the startup algorithm for DACAPO as a network of timed automaton and used the UPPAAL model checker to verify its safety and timeliness properties [53]. They did not consider any transient or permanent communication faults in their model. Though disturbances on the bus may cause an initial loss of synchronization, no fault can occur during the execution of the startup algorithm. Their model is restricted to only four nodes, and the authors apprehended that it would be very challenging to extend the same analysis to more than four nodes.

Pike and Johnson carried out the verification of the reintegration algorithm [54] for NASA Langley's SPIDER (Scalable Processor-Independent Design for Electromagnetic Resilience) architecture [52]. SPIDER uses a star configuration with a centralized hub to manage redundancy and provide fault-tolerance. The reintegration algorithm is actually a variant of the startup algorithm, where a node which loses synchronization with other nodes due to some transient faults, tries to resynchronize itself with a set of synchronized nodes. The authors perform a timeout-based continuous time modeling of the reintegration algorithm using SAL and verify a couple of safety properties using k-induction. The k-induction proof requires a number of supporting lemmas. They adopt a compositional proof strategy where the overall verification is decomposed into the verification of the constituent modes. Their fault assumption follows the "Majority" property which says that the majority of the nodes that are synchronized should not show any faulty behavior. The faulty nodes may show arbitrary Byzantine behavior. The algorithm has been verified for four nodes with one re-integrator, and, at most, one of the three synchronized nodes can be faulty to satisfy the Majority property.

IX. Discussion

Time-triggered architectures invoke a lot of interest in engineers in CPS industries, such as avionics and automotive industries. These architectures employ complex algorithms for real-time and distributed computations. As the applications of these architectures are safety-critical, rigorous verification of these algorithms assumes importance. While formal verification techniques are known to suffer from lack of scalability, they have been successfully applied to small scale systems to get useful insight about the design of distributed algorithms. Though the comprehensive verification of a large scale system like a time-triggered architecture is beyond our reach with the current formal methods technologies, the efforts on the verification of startup algorithms demonstrate that formal method based verification techniques can be very useful to acquire confidence on small but critical components of safety-critical systems.

Among all the algorithms used in time-triggered architectures, the startup algorithm has received the most attention from the research community. In this paper, we surveyed the formal verification efforts carried out for checking the correctness of startup algorithms for different time-triggered architectures. We briefly mention here the verification works carried out for other algorithms used in time-triggered architectures. The mechanical verification of fault-tolerant clock synchronization algorithms was carried out independently by Shankar [56], Rushby [57] and Schwier and von Henke [58]. Pfeifer et al. verified the clock synchronization algorithm for TTP/C [31]. Pfeifer undertook the formal verification of a fault-tolerant group membership algorithm for TTP/C architecture [32]. The formal verification of a bus guardian window timing for TTP/C architecture was carried out by Rushby [33]. Recently, Rosset et al.verified a group membership algorithm for the FlexRay protocol using UPPAAL model checker [59].

There is a lot of scope to carry out further research in this field. In the calendar-based model of TTA [5] an exhaustive fault simulation can be undertaken. In [7], the clockless model of the TTA startup algorithm does not consider the fault-tolerant aspects of the algorithm. The model should be extended incorporating the fault-tolerant aspects of the algorithm. Also, an exhaustive fault simulation should be carried out in this work. In the case of the FlexRay startup algorithm verification [8], a rigorous study of the robustness of the startup algorithm in more difficult scenarios is yet to be undertaken. Such exhaustive fault simulation may require the design of appropriate guardian instances. Also a calendar-based model of FlexRay can bring out some interesting features in the verification process. Finite state modeling and the verification of FlexRay and TTCAN protocols by using a clockless modeling technique may be good experiments to judge the effectiveness of the clockless modeling technique in verifying time-triggered protocols. In this regard, one needs to find out the extent to which digitization properties could be utilized in verifying these time-triggered architectures. Further, there are a few other time-triggered architectures, e.g., LIN [60] and SAFEbus [61], and the modeling and verification of the startup algorithms for these architectures may lead to interesting work. The modeling and verification techniques described in this paper would certainly enable one to get a sufficient insight into the analysis of a startup algorithm for any time-triggered architecture. Moreover, any algorithm which involves time-triggered operations and handles fault tolerance, can be verified using similar techniques. Finally, though theorem proving techniques have been used for other algorithms for time-triggered architectures, it has never been tried for startup algorithms. Although theorem proving lacks full automation, it may be possible to prove useful properties of startup algorithms for any number of nodes using theorem proving techniques, which is yet to be undertaken.

In the automotive and avionics industries, the current practices of verification of embedded control software depend primarily upon dynamic techniques like testing and simulation. The software development follows model-based techniques using modeling environments like Rhapsody [62] and Simulink/Stateflow [63]. These environments provide extensive facilities for model simulation, using which models are validated by simulating them over typical

TABLE I
SUMMARY OF THE MODEL CHECKING EFFORTS OF THE STARTUP ALGORITHMS

Startup Algorithm	Modeling and Verification Done by	Model Checker Used	Model of Time	Fault Modeling Approach	Correctness Properties Verified	Performance of the Model
TTA	Steiner et. al. [6]	SAL	Discrete Time Modeling	Exhaustive fault simulation for single fault hypothesis	Safety, Liveness, Timeliness	Models examined for 3, 4, 5 and 6 nodes and 2 channels
	Dutertre and Sorea [5]	SAL	Calendar-based continuous time modeling	Single fault hypothesis (one node can be faulty)	Safety	Modeling done for up to 10 nodes
	Saha, Misra and Roy [7]	SPIN	Calendar-based discrete time clockless modeling	Fault free	Safety, Liveness	Modeling done for up to 10 nodes
FlexRay	Steiner [8]	SAL	Discrete Time modeling	Single fault hypothesis (one node or one channel can be faulty)	Safety, Liveness, Timeliness	Modeling done for 4 nodes
TTCAN	Saha, Roy and Chakraborty [9]	SAL	Synchronous Calendar-based Continuous time modeling	Any number of crash failure	Safety, Liveness	Modeling done for upto 9 nodes
DACAPO	Lönn and Pettersson [53]	UPPAAL	Timed Automata modeling	Fault free	Safety, Timeliness	Modeling done for 4 nodes
SPIDER	Pike and Johnson [55]	SAL	Timeout-based continuous time modeling	Number of faulty nodes satisfies majority property	Safety	Modeling done for 4 nodes

scenarios and input sequences. The final system includes, besides applications, the networking and operating system infrastructure. To make sure that the infrastructure is free of bugs, system level testing is employed. Application level simulation and system level testing cannot, however, provide absolute correctness guarantee. Rigorous verification methods like the one discussed in this paper would be required to enhance confidence in the systems being developed. Moreover, the application of rigorous methods like formal techniques is slowly gaining acceptance in these industries. Many safety standards in avionics (e.g., IEC 61508 [64], DO 178B [65]) and the draft automotive software standard like ISO 26262 [66] recommend the use of formal methods. The verification approaches discussed in this survey report would certainly aid those activities.

Acknowledgement A significant part of this work was carried out when the first and the second authors were with Honeywell Technology Solutions Lab, Bangalore. The authors appreciate many helpful comments received from Prahladavaradan Sampath. The authors thank Wilfried Steiner for his help with the improvement of the presentation of the paper. Thanks are also due to anonymous reviewers for providing many useful remarks and suggestions.

REFERENCES

[1] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," *Proceedings of the IEEE (special issue on CPS)*, vol. 100, no. 1, pp. 13–28, 2012.
[2] E. A. Lee, "CPS foundations," in *Proc. Design Automation Conference (DAC)*, 2010, pp. 737–742.
[3] H. Kopetz, "Real-Time Systems: Design Principles for Distributed Embedded Applications," *The Kluwer International Series in Engineering and Computer Science*, 1997.
[4] ——, "The Time-Triggered Model of Computation," in *Proceedings of RTSS*, 1998, pp. 168–177.
[5] B. Dutertre and M. Sorea, "Modeling and Verification of a Fault-Tolerant Real-Time Startup Protocol Using Calendar Automata," in *Proceedings of FORMATS/FTRTFT*, 2004, pp. 199–214.
[6] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer, "Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration to Exhaustive Fault Simulation," in *Proceedings of DSN*, 2004, pp. 189–198.
[7] I. Saha, J. Misra, and S. Roy, "Timeout and Calendar Based Finite State Modeling and Verification of Real-Time Systems," in *5th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, ser. LNCS, vol. 4762, 2007, pp. 284–299.

[8] W. Steiner, “Model-Checking Studies of the FlexRay Startup Algorithm,” Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Research Report 57/2005, 2005.

[9] I. Saha, S. Roy, and K. Chakraborty, “Modeling and Verification of TTCAN Startup Protocol Using Synchronous Calendar,” in *IEEE Conference on Software Engineering and Formal Methods (SEFM)*, 2007, pp. 69–79.

[10] W. Steiner and H. Kopetz, “The Startup Problem in Fault-Tolerant Time-Triggered Communication,” in *Proceedings of DSN*, 2006, pp. 35–44.

[11] C. Temple, “Avoiding the babbling-idiot failure in a time-triggered communication system,” in *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, 1998, pp. 218–227.

[12] H. Kopetz and G. Grünsteidl, “TTP - A Protocol for Fault-Tolerant Real-Time Systems,” *IEEE Computer*, vol. 27, no. 1, pp. 14–23, January 1994.

[13] SpecTTP/C, “Specification of the TTP/C Protocol (version 0.6p0504),” May 2001.

[14] SpecFlexRay, “FlexRay Communication System - Protocol Specification - Version 2.0,” FlexRay Consortium, Tech. Rep., 2004.

[15] ISO 11898: Road Vehicles, *Interchange of Digital Information - Part 1: Controller Area Network (CAN) for High-Speed Communication*, 2003.

[16] ——, *Interchange of Digital Information - Part 2: High-Speed Medium Access Unit*, 2003.

[17] W. Steiner and M. Paulitsch, “The Transition from Asynchronous to Synchronous System Operation: An Approach for Distributed Fault-Tolerant Systems,” in *Proceedings of ICDCS*, 2002, pp. 329–336.

[18] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther, “Time-Triggered Communication on CAN (Time-Triggered CAN-TTCAN),” Robert Bosch GmbH.

[19] H. Hopkins, “Fit and Forget Fly-by-wire,” *Flight International*, pp. 89–92, December 1988.

[20] J. Rushby, “Bus Architectures for Safety-Critical Embedded Systems,” in *Proceedings of EMSOFT*, 2001, pp. 306–323.

[21] G. Bauer, H. Kopetz, and P. Puschner, “Assumption Coverage under Different Failure Modes in the Time-Triggered Architecture,” in *Proceedings of International Conference on Emerging Technologies and Factory Automation*, 2001, pp. 333–341.

[22] A. Ademaj, G. Bauer, H. Sivencrona, and J. Torin, “Evaluation and Fault Handling of the Time-Triggered Architecture with Bus and Star Topology,” in *Proceedings of DSN*, 2003, pp. 123–132.

[23] G. Bauer, H. Kopetz, and W. Steiner, “The Central Guardian Approach to Enforce Fault Isolation in a Time-Triggered System,” in *Proceedings of 6th International Symposium on Autonomous Decentralized Systems*, 2003, pp. 37–44.

[24] T. Yokogawa, T. Tsuchiya, and T. Kikuno, “Automatic Verification of Fault-Tolerance Using Model Checking,” in *Proceedings of 2001 Pacific Rim International Symposium on Dependable Computing*, 2001, pp. 95–102.

[25] L. Lamport, R. Shostak, and M. Pease, “The Byzantine General Problem,” *ACM Transaction on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[26] R. Alur and D. L. Dill, “A Theory of Timed Automata,” *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[27] B. Dutertre and M. Sorea, “Timed Systems in SAL,” Computer Science Laboratory, SRI International, Tech. Rep., 2004.

[28] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[29] E. M. Clarke, O. Grümberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[30] S. Owre, J. M. Rushby, and N. Shankar, “PVS: A prototype verification system,” in *Proceedings of International Conference on Automated Deduction*, 1992, pp. 748–752.

[31] H. Pfeifer, D. Schwier, and F. W. von Henke, “Formal Verification for Time-Triggered Clock Synchronization,” in *Proceedings of Dependable Computing for Critical Applications*, 1999, pp. 207–226.

[32] H. Pfeifer and U. Ulm, “Formal Verification of the TTP Group Membership Algorithm,” in *Proceedings of FORTE/PSTV*, 2000, pp. 3–18.

[33] J. Rushby, “Formal Verification of Transmission Window Timing for the Time-Triggered Architecture,” Computer Science laboratory, SRI International, Tech. Rep., 2001.

[34] G. J. Holtzman, *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003.

[35] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueβ, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari, “An Overview of SAL,” in *Proceedings of the Fifth NASA Langley Formal Methods Workshop, NASA Langley Research Center*, 2000, pp. 187–196.

[36] L. M. de Moura, S. Owre, H. Rueß, J. M. Rushby, N. Shankar, M. Sorea, and A. Tiwari, “SAL 2.” in *Proceedings of CAV*, 2004, pp. 496–500.

[37] K. G. Larsen, P. Pettersson, and W. Yi, “Uppaal in a Nutshell,” *Int. Journal on Software Tools for Technology Transfer*, vol. 1, pp. 134–152, 1997.

[38] O. Lichtenstein and A. Pnueli, “Checking That Finite State Concurrent Programs Satisfy Their Linear Specification,” in *Proceedings of POPL*, 1985, pp. 97–107.

[39] E. M. Clarke, E. A. Emerson, and A. P. Sistla, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications,” *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[40] E. M. Clarke, O. Grümberg, and M. C. Browne, “Reasoning about Networks with Many Identical Finite-State Processes,” in *Proceedings of PODC*, 1986, pp. 240–248.

[41] S. Aggarwal, C. Courcoubetis, and P. Wolper, “Adding Liveness Properties to Coupled Finite-State Machines,” *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 2, pp. 303–339, 1990.

[42] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, H. Zheng, and W. Visser, “Tool-Supported Program Abstraction for Finite-State Verification,” in *Proceedings of ICSE*, 2001, pp. 177–187.

[43] M. C. Browne, E. M. Clarke, and O. Grümberg, “Characterizing Finite Kripke Structures in Propositional Temporal Logic,” *Theoretical Computer Science*, vol. 59, no. 1-2, pp. 115–131, 1988.

[44] E. M. Clarke, O. Grümberg, M. Minea, and D. Peled, “State Space Reduction Using Partial Order Techniques,” *STTT*, vol. 2, no. 3, pp. 279–287, 1999.

[45] G. J. Holzmann, “An Analysis of Bitstate Hashing,” *Formal Methods in System Design*, vol. 13, no. 3, pp. 289–307, 1998.

[46] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu, “Bounded Model Checking Using Satisfiability Solving,” *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001.

[47] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, “Symbolic model checking for real-time systems,” in *Proceedings of Seventh Annual IEEE Symposium on Logic in Computer Science*, 1992, pp. 394–406.

[48] J. Rushby, “Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification,” in *Proceedings of CAV*, 2000, pp. 508–520.

[49] Z. Manna and A. Pnueli, “Temporal Verification Diagrams,” in *Proceedings of International Symposium on Theoretical Aspects of Computer Software*, 2004, pp. 726–765.

[50] H. Hartwich, B. Müller, T. Führer, and R. Hugel, “Timing in the TTCAN Network,” Robert Bosch GmbH.

[51] B. Rostamzadeh, H. Lonn, R. Snedsbol, and J. Torin, “[DACAPO]: A Distributed Computer Architecture for Safety-Critical Control Applications,” in *Proceedings of the Intelligent Vehicles ’95 Symposium*, 1995, pp. 376–381.

[52] P. S. Miner, “Analysis of the SPIDER Fault-Tolerance Protocols,” in *Proceedings of Fifth NASA Langley Formal Methods Workshop*, 2000.

[53] H. Lönn and P. Pettersson, “Formal verification of a TDMA protocol start-up mechanism,” in *Fault-Tolerant Systems, 1997. Proceedings., Pacific Rim International Symposium on*, 1997, pp. 235–242.

[54] W. Torres-pomales, M. R. Malekpour, and P. S. Miner, “A Fault-Tolerant Broadcast Communication System,” NASA, Tech. Rep. NASA/TM-2005-213540 ROBUS-2, 2005.

[55] L. Pike and S. D. Johnson, “The formal verification of a reintegration protocol,” in *Proceedings of EMSOFT*, 2005, pp. 286–289.

[56] N. Shankar, “Mechanical Verification of a Generalized Protocol for Byzantine Fault-Tolerant Clock Synchronization,” in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 1992, pp. 217–236.

[57] J. Rushby, “A Formally Verified Algorithm for Clock Synchronization under a Hybrid Fault Model,” in *Thirteenth ACM Symposium on Principles of Distributed Computing*, 1994, pp. 304–313.

[58] D. Schwier and F. von Henke, “Mechanical Verification of Clock Synchronization Algorithms,” in *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, 1998, pp. 262–271.

[59] V. Rosset, P. Souto, and F. Vasques, “Formal Verification of a Group Membership Protocol Using Model Checking,” in *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, 2007, pp. 471–488.

[60] “Local Interconnect Network (LIN),” 2000, http://www.lin-subbus.org/.

[61] K. Hoyme and K. Driscoll, “SAFEbus^{TM},” in *Proceedings of 11th AIAA/IEEE Digital Avionics Systems Conference*, 1992, pp. 68–73.

[62] Telelogic, “Rhapsody: UML Model-Driven Development,” http://modeling.telelogic.com.

[63] The Mathworks, “MATLAB and Simulink,” http://www.mathworks.com.

[64] H. Gall, “Functional Safety IEC 61508 / IEC 61511 the Impact to Certification and the User,” in *The 6th ACS/IEEE International Conference on Computer Systems and Applications(AICCSA 2008)*, 2008, pp. 1027–1031.
[65] *RTCA Inc., Document RTCA/DO-178B*, Federal Aviation Administration, Jan. 11, 1993, advisory Circular 20-115B.
[66] “ISO/FDIS 26262-1: Road vehicles – Functional safety – part 1: Vocabulary,” 2010, available at http://www.iso.org/iso/catalogue_detail.htm?csnumber=43464.

Indranil Saha is an Assistant Professor in the Department of Computer Science and Engineering at Indian Institute of Technology, Kanpur. He was a postdoctoral researcher affiliated with the EECS department at University of California Berkeley and the CIS department at University of Pennsylvania during 2013-2015. He obtained his Ph.D. degree from the Department of Computer Science at University of California Los Angeles in 2013. His research interest is application of formal methods to embedded and cyber-physical systems and robotics.

Suman Roy is a senior research scientist with Infosys Ltd, Bangalore. He holds a B.E. degree from Jadavpur University, Kolkata, an M.E. degree from Indian Institute of Science (IISc), Bangalore and a Ph.D. degree in Computer Science and Automation also from IISc. Prior to Joining Infosys, he was affiliated with the research labs of Honeywell and Mercedes-Benz. His research interests include formal methods in software engineering, semantic technologies, semantic extraction and computer/network security.

S. Ramesh is a technical fellow at General Motors R&D. He is also an adjunct professor at International Institute of Information Technology, Bangalore. Before joining to GM R&D, he was a professor at Indian Institute of Technology, Bombay for sixteen years. He obtained his Ph.D. degree in Computer Science from the Indian institute of Technology, Bombay in 1997. His research interest is validation and verification of embedded software.