Modeling and Verification of TTCAN Startup Protocol Using Synchronous Calendar

Indranil Saha and Suman Roy
HTS(Honeywell Technology Solutions) Research
151/1 Doraisanipalya, Bannerghatta Road
Bangalore 560 076, India
{indranil.saha, suman.roy}@honeywell.com

Kuntal Chakraborty*
Indian Statistical Institute
203 Barrackpore Trunk Road
Kolkata 700108, India
mtc0502@isical.ac.in

Abstract

We describe the modeling and verification of TTCAN startup protocol using SAL model checker. For the modeling purposes we propose a new modeling framework called Synchronous Calendar which can be seen as an adaptation of Calendar based models introduced by Duterte and Sorea. A Synchronous Calendar can express dense time systems without relying on continuously varying clocks and supports synchronous message transmission. We capture both fault-free and fault-tolerant aspects of startup algorithm of TTCAN in two different models and verify the safety and liveness properties for them. Our verification technique relies on induction and abstraction methods which are supported by SAL model checker. To our knowledge this is the first work towards a formal analysis of TTCAN startup protocol.

1 Introduction

In modern day automobiles the communication between micro-controllers, sensors and actuators is widely based on event triggered communication on CAN [9] protocol. The arbitration mechanism of this protocol ensures that all messages are transferred according to the priority of their identifiers and the message with the highest priority is always delivered. For the next generation vehicles some mission critical subnetworks, *e.g.*, *x-by-wire systems (xbws)* [5] will require additional deterministic behavior in communication during service. Even at maximum bus load, transmission of all safety related messages must be guaranteed. One way to solve this issue using CAN is to extend the standard CAN protocol to a time triggered protocol (TTCAN) [6]. In TTCAN, the communication is based on the periodic transmission of a *reference message* by a special node called *time master*. The period between two consecutive reference messages is called the *basic cycle* (Figure 1). This allows to introduce a global network time across the system with high precision. Based on this time different messages are assigned to different time windows within a basic cycle. A big advantage of TTCAN compared to classical scheduling systems is the possibility to transmit event triggered messages in certain *arbitrating time windows* as well.

TTCAN is built on the top of CAN protocol using a Time-Triggered Architecture (TTA). In time triggered architectures all system activities are initiated by the progression of time [11]. All nodes should be synchronized in time and every activity in the network is time stamped using the global time as defined by the time master. The message-schedule can be determined prior to the start of the system because all messages are allocated time on the bus at the design level. For proper operation, time-triggered architectures depend on some basic algorithms, viz. *bus guardian window timing*, *group membership*, *clique avoidance*, *non-blocking write*, *clock synchronization* and *startup*. Among all these algorithms, startup algorithms are exciting targets for formal verification as the nodes of the system interact in interesting ways. In this work we model and verify the fault-free and fault-tolerant startup algorithms for TTCAN using the model checker SAL [12].

SAL (Symbolic Analysis Laboratory) is a framework for the specification and analysis of concurrent systems. It consists of the SAL language [2], which provides notations for specifying state machines and their prop-

*A part of the work was done when the author was a summer intern at HTS, Bangalore during May-July'06.

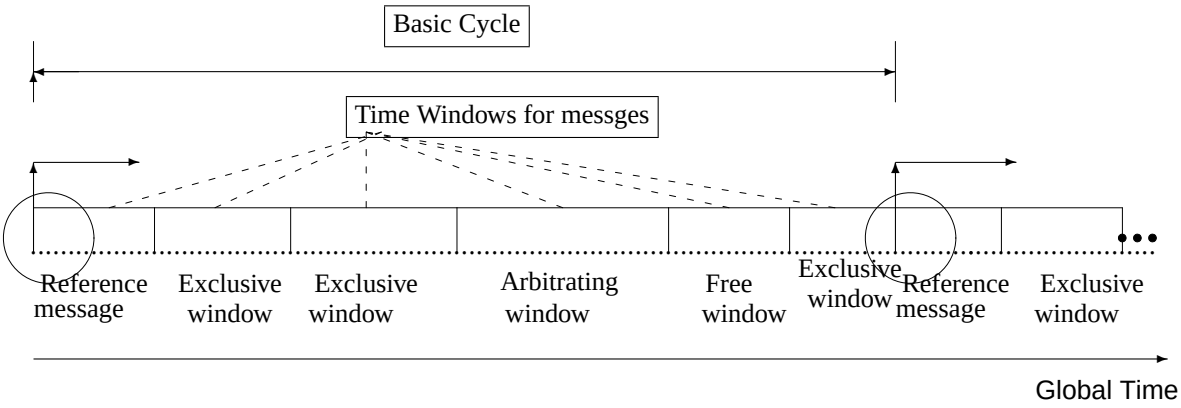

Figure 1. State diagram of a general node for fault-free startup

erties, and the SAL system [12] that provides model checkers and other tools for analyzing properties of state machine specifications written in SAL. This tool helps analyzing systems that combine real-valued and discrete state variables and can then apply to real-time systems with a dense time model. SAL is a generalist tool, intended for the modeling and verification of discrete transition systems, and not for systems with continuous dynamics. As a consequence, existing models such as timed automata [1], which employ continuous clocks, do not fit the SAL framework very well.

To overcome this problem Dutertre and Sorea [3, 4] proposed a new class of timed transition systems that use dense time, but do not require continuously varying state variables, and are then better suited to SAL. They borrow the concepts of these models from the event calendars [10, 14] used in computer simulation of discrete event systems and propose two modeling frameworks: Timeout based model which can be used for modeling system with no message passing and Calendar based model which is suitable for modeling systems where message transmission is asynchronous with bounded delay. As message passing is synchronous (no transmission delay between the sender and the receiver) in TTCAN [7] none of these frameworks can be directly used for modeling its startup algorithm. We propose a new modeling framework called Synchronous Calendar, which can capture synchronous message transmission.

The remainder of the paper is organized as follows. In section 2 we mention some related works carried out in the past on modeling and verification of startup protocols for some time-triggered architectures. In section 3 we discuss fault-free and fault-tolerant startup algorithms of TTCAN. In section 4 we introduce Synchronous Calendar as a new modeling framework. Modeling of fault-free and fault-tolerant startup protocol by using Synchronous Calendar based modeling framework have been described in section 5. Interesting properties relevant to these startup algorithms are discussed in section 6 and induction and abstraction based verification methodologies have also been described. Finally, we conclude in Section 7.

2 Related Work

Modeling and verification of startup protocols for time-triggered architectures is a very active field of research in the recent years, as witnessed by a number of works in this area [3, 13, 16, 17, 18]. There are two pieces of work on model-checking TTA startup algorithm in the literature. The first one has been carried out by Steiner et. al. [17] using SAL model checker. This is a discrete time modeling which uses an integer counter to model the propagation of time. Exhaustive fault simulation has been undertaken in the work. The verification of 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. Dutertre and Sorea [3] carried out continuous time modeling of TTA startup using SAL model checker. In this work, continuous time dynamics have been captured by using Calendar based model, where a global data structure *Calendar* has been used to capture message transmission delays. They have modeled the protocol with an active hub that is assumed to be reliable, but a single node may be Byzantine faulty, and can attempt to broadcast arbitrary frames at any time. Model-checking of FlexRay startup algorithm has been carried out by Steiner [16] using SAL model checker. In this work, he has done discrete time modeling of the FlexRay

startup algorithm, the time model being similar to the one in [17]. Pike and Johnson present a formal verification of the SPIDER Reintegration Protocol [13] using SAL model checker. Reintegration protocol is similar to Startup protocol, the only difference is unlike Startup protocol, Reintegration protocol does not run during system powerup, rather a non-faulty node outside the operational clique joins the operational clique through Reintegration Protocol. Modeling and verification of ASCB-D startup algorithm has been carried out by Weininger and Cofer [18] using Spin model checker [8]. They have introduced an explicit numerical time model, and combined time-modeling capability and the message-transmission capability in a process called *environment*. To our knowledge no work has been done on the modeling and verification of startup algorithm of TTCAN startup algorithm which has a potential of being a good case study.

3 Startup Protocol of TTCAN

In this section we shall briefly describe the fault-free and fault-tolerant startup algorithms for TTCAN. For correct operation, all the TTCAN nodes should be synchronized in time. When the system powers up, all the nodes start in normal CAN mode of operation. The startup algorithm establishes initial synchronization in the system. Among all the nodes a subset of nodes can participate in the process, and these nodes are called *potential time masters*. All the other nodes are called *general nodes*. There are more than one potential time masters for providing fault-tolerance in the system and there is a strict order relationship of identifiers of the potential time masters. According to their identifiers their priorities are determined. The lower the identity the higher is the priority of the potential time master. The objective of the startup algorithm is to establish the highest priority potential time master as the active time master, and this active time master takes over the responsibility of maintaining synchronization in the system by sending reference message periodically. First, a potential time master checks whether the bus is empty and if there is a reference message on the bus. If the above is not true, the potential time master sends a reference message with its identifier. If two potential time masters attempt to send the reference message at the same time, the arbitration mechanism of CAN resolves this contention. Whenever a reference message with higher priority is received by any potential time master it synchronizes with the existing time master. If a reference message with lower priority is received then the potential time master first synchronizes with the existing time master and then tries to become time master by sending its own reference message at the start of the next basic cycle. If more than one potential time masters participate in this arbitration, the potential time master with the highest priority wins the arbitration. The protocol ensures that under error free condition the potential time master with the highest priority eventually becomes the active time master.

During the startup process the failure of the time master is recognized by detection of a missing reference message within a short latency. The latency is realized by a timeout. When this timeout is reached a potential time master starts sending the reference message with its global time as content. The bitwise arbitration of the standard CAN protocol decides the next time master among competing potential time masters. Subsequently the functionality of the time master is reestablished.

4 A Synchronous Calendar-based Model

Dutetre and Sorea [3] introduced timeout based model where state variables include current time t and a finite set T of timeouts. In a real-time system with n processes, the system can be modeled with n timers, where each timer denotes the next discrete transition for a process. This kind of modeling technique is very efficient to model systems where all the discrete transitions depend on some timeouts.

Timeout based modeling technique is not applicable to systems where some discrete transitions do not occur due to timeouts, rather occur on receiving messages. To model interaction through message passing event calendars are introduced in [3]. A calendar is defined as a finite set of the form $C = \{< e_1, t_1 >, < e_2, t_2 >, \ldots ,< e_n, t_n >\}$, where e_i is an event which is scheduled to occur at time t_i. These calendars are advantageous to model a system where the processes communicate asynchronously, and it is known when a message will reach the destination. When a process sends a message, it is stored in the calendar along with the information when it is scheduled to be delivered to the receiver. When a message is received, the corresponding entry is removed from the calendar.

While calendar-based modeling is appropriate to model systems where communication is asynchronous with bounded message propagation delay, it cannot be applied to model synchronous communication directly. On the other hand, timeout based model is not adequate in such situations where some discrete transitions are event triggered. For example, if some discrete transition occurs due to receiving some message, it cannot be cap-

tured by mere timeout based model. To model this kind of system it is required to keep track of which processes are to receive a particular message and whether the same is received by the receiver, along with the information of the timeouts for individual processes. For example, in TTCAN the active time master sends the reference message and this message is received by the nodes who are ready to receive it. As signal propagation time is considered to be negligible in TTCAN [7], there is no need to consider the time of delivery of the message, but the sender and receivers of the messages should be recorded properly. This can be done by using a flag for every node present in the network. When a message is sent only the flags corresponding to the valid receivers are made true. When a receiver receives the message, the corresponding flag is made false.

To capture synchronous communication we introduce a data structure called *Synchronous Calendar*. A Synchronous Calendar can be formally defined as $SC = < s, F >$, where s is the sender of the message, and F is the set of boolean variables, where $|F| = n$ is the number of nodes in the system. For a node i, the corresponding flag $f_i \in F$ is set to true, if sender s wants to send its message to i. As we are dealing with synchronous communication, only one entry in SC is possible at a particular time. When there is no entry in SC, we say that it is EMPTY.

We shall now describe how transitions in the systems are guided by timeouts and Synchronous Calendar.

- *Initial State:* In all initial states σ_0, we have $\sigma_0(t) \leq \min(\sigma_0(T))$ and $\sigma_0(SC) = EMPTY$.
- *Time Progress:* In a state σ, time can progress if and only if $\sigma(t) < \min(\sigma(T))$ and $\sigma(SC) = EMPTY$. A time progress transition updates t to $\min(\sigma(T))$ leaving all other state variables unchanged.
- *Discrete transitions:* They can be enabled in a state σ provided $\sigma(t) = \min(\sigma(T))$ or $\sigma(SC) \neq EMPTY$, and they must satisfy the following rules:
  - $\sigma'(t) = \sigma(t)$.
  - $\forall y \in T$ we have $\sigma'(y) = \sigma(y)$ or $\sigma'(y) > \sigma'(t)$
  - $\sigma(SC) \neq EMPTY \Rightarrow \sigma'(SC) = EMPTY$.
  - $\sigma(SC) = EMPTY$ and $\exists y \in T$ such that $\sigma(y) = \sigma(t) \Rightarrow \sigma'(y) > \sigma'(t)$.

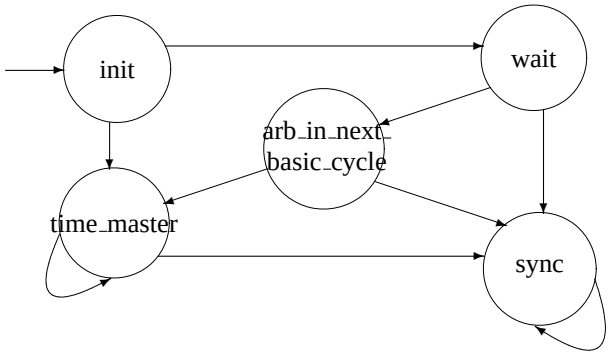

Figure 2. State diagram of a node for fault-free startup

The above rules imply that time cannot progress if a synchronous communication is enabled. Discrete transition may occur due to synchronous communication or due to a timeout. If both can be enabled at the same time then the discrete transition for synchronous communication will get the priority. If the discrete transition is due to timeout, then the corresponding timeout has to be increased to a future value.

In some cases it may not be required to increase the timeout, for the very next transition for the node is due to a synchronous communication. For example, in TTCAN when init timeout occurs for a potential time master, it is obvious that its next discrete transition will occur after getting a reference message from the time master. So apparently it is not required to increase the timeout for the particular node. But the rules for discrete transition say that if the discrete transition is due to timeout, the timeout has to be increased to a future value. This is required to ensure that $\sigma(t) \leq min(\sigma(T))$ is an invariant. In this case, this timeout would be a dummy timeout, and should be set carefully, so that it is not scheduled before the discrete transition due to next synchronous communication.

5 Modeling of TTCAN startup Protocol in SAL

We shall describe two models for TTCAN startup: fault-free and fault-tolerant startup model.[1]

5.1 Fault-free Startup Model

The state diagram of a TTCAN node executing startup algorithm in fault-free scenario is depicted in

[1]Interested readers are requested to contact the authors to obtain the SAL models.

Figure 2. The state diagram captures behaviors of both the potential time masters and the general nodes. Initially, all the potential time masters are in the *init* state where they can remain upto an arbitrary amount of time which is captured as a timeout for this particular state. When this init timeout occurs for a potential time master, it checks whether the bus is empty. If it is so, the potential time master moves to *time_master* state, and sends a reference message simultaneously. Henceforth, we shall call the node in *time_master* state the *current time master* or *active time master*. Timeouts of two potential time masters in *init* state may occur at the same time. In this case, arbitration decides which node will be the active time master. The winner potential time master moves to *time_master* state, all the loser potential time masters move to *wait* state. If a node in the *init* state receives a message from the current time master, it discards the message. If a timeout occurs for a node in the *init* state and it finds that bus is not empty then it moves to the *wait* state and waits for the next reference message. From *wait* state two transitions are possible. When a potential time master in the *wait* state receives a reference message, it checks whether its priority is lower than that of the current time master. If the priority of the current time master is greater than its own priority, then it moves to *sync* state. Otherwise the node moves to *arb_in_next_basic_cycle* state. The nodes in the *arb_in_next_basic_cycle* state are synchronized with the current time master. They define their new timeout to be current time plus basic cycle. The timeouts for the nodes in *arb_in_next_basic_cycle* state and *time_master* state occur at the same point of time. Now the current time master checks if it should remain in its present state or move to *sync* state. If there are some nodes in the *arb_in_next_basic_cycle* state, then current time master loses the arbitration and moves to *sync* state. A node in the *arb_in_next_basic_cycle* state checks whether it has won the arbitration. In case it wins, it moves to *time_master* state. Otherwise it moves to *sync* state.

To capture this synchronous behavior of communication in the start-up process, we use Synchronous Calendar. This is called `reference_message_calendar` in the model. It holds the id of the sender of reference message, and boolean flags for all potential time masters and general nodes.

```
SYNCHRONOUS_CALENDAR:TYPE =
  [# master_id : IDENTITY1,
     not_received_flag1: ARRAY IDENTITY1 OF BOOLEAN,
     not_received_flag2: ARRAY IDENTITY2 OF BOOLEAN
  #];
ref_message_calendar: SYNCHRONOUS_CALENDAR;
```

`IDENTITY1` and `IDENTITY2` denote the IDs of potential time masters and general nodes respectively.

We have modeled all the potential time masters as a parameterized module. INPUT, OUTPUT, LOCAL, GLOBAL variables for a potential time master have been specified as follows.

```
potential_time_master[i: IDENTITY1]: MODULE =
  BEGIN
    INPUT  time: TIME
    OUTPUT timeout: TIME
    OUTPUT pc: PC
    GLOBAL ref_message_calendar: SYNCHRONOUS_CALENDAR
    GLOBAL bus_traffic: BOOLEAN
    GLOBAL winner_id: IDENTITY1
```

A potential time master reads the current `time` via an input state variable and exports output variables corresponding to its local `timeout` and its current state `pc`. The variables `time` and `timeout` are of type TIME which is actually of a REAL type. A potential time master has access to the global structure `ref_message_calendar`, boolean variable `bus_trafic` that indicates if there is a node in the *time_master* state, and another boolean variable `winner_id` that denotes the winner of an arbitration.

In the INITIALIZATION section, necessary OUTPUT and GLOBAL variables are initialized as follows:

```
INITIALIZATION
 pc = init;
 timeout IN { x: TIME | time<x AND x<basic_cycle_time};
 ref_message.calendar = EMPTY_CALENDAR;
```

EMPTY_CALENDAR is a special value of the `ref_message_calendar` with the sender id equal to 0 along with all the flags set to false. It denotes that no reference message has been sent. In TRANSITION section of the module, we specify all the possible transition of a potential time master by a set of guarded commands. The usefulness of Synchronous Calendar for the reference message is best illustrated in the case when some nodes are in the *wait* state and some are in the *init* state, and there is already an active time master. If a node in the *init* state sees that its `not_received_flag` is true, it neglects the reference message by setting its `not_received flag` to false.

```
init_to_init:
  pc = init AND
  ref_message_calendar.not_received_flag1[i] = TRUE -->
    pc' = init;
    ref_message_calendar' = ref_message_calendar
                WITH .not_received_flag1[i] := FALSE
```

When a potential time master moves to *wait* state, there is already a potential time master in the *time_master* state, which has already sent a reference message. But the nodes in the *wait* state discarded that

reference message when they were in the *init* state. After coming in the *wait* state when a potential time master sees its `not_received_flag` in the calendar as true, it understands that a new reference message has been sent. It moves to *sync* state or *arb_in_next_basic_cycle* state depending on whether current time master's priority is greater or less than its own priority. The transition from *wait* to *sync* state is presented below.

```
[] wait_to_sync:
  pc = wait AND
  ref_message_calendar.not_received_flag1[i] = TRUE
  AND ref_message_higher_priority?(i) = TRUE -->
    pc' = sync;
    timeout' = time + basic_cycle_time;
    ref_message_calendar' = ref_message_calendar
                WITH .not_received_flag1[i] := FALSE
```

In the state diagram, only two transitions are possible for a general node. Initially, all the general nodes are in *init* state where they stay upto an arbitrary amount of time. Then they move to *wait* state, where they wait for a reference message from the active time master. Once it gets a reference message, it becomes synchronized with the time master and moves to *sync* state. The module for general nodes has been designed in the similar way as that of potential time masters. The guarded transition for a general node from *wait* state to *sync* state is presented below:

```
[ ] wait_to_sync:
  pc=wait AND
  ref_message_calendar.not_received_flag2[i] = TRUE -->
    pc' = sync;
    timeout' = time + basic_cycle_time;
    ref_message_calendar' = ref_message_calendar
                WITH .not_received_flag2[i] := FALSE;
```

To capture the advancement of time properly, we have defined a `clock` module, which takes the timeouts of individual potential time masters and general nodes as INPUT, and outputs the updated time. It uses two functions `time_can_advance` and `is_next_event`. The function `time_can_advance` returns true if time progression is possible. The function `is_next_event` is used to find out a suitable time point in future where `time` can advance to. The `clock` module along with these two functions is presented below.

```
time_can_advance(time: TIME, time_out1: TIMEOUT_ARRAY1,
 time_out2: TIMEOUT_ARRAY2): BOOLEAN=
   FORALL(i: IDENTITY1): time < time_out1[i]
   AND FORALL(j: IDENTITY2): time < time_out2[i]
   AND FORALL(i: IDENTITY1):
   ref_message_calendar.not_received_flag1[i] = FALSE
   AND FORALL(j: IDENTITY2):
   ref_message_calendar.not_received_flag2[j] = FALSE;

is_next_event(t: TIME, time_out1: TIMEOUT_ARRAY1,
               time_out2: TIMEOUT_ARRAY2): BOOLEAN =
   FORALL(i:IDENTITY1): t <= time_out1[i]
   AND FORALL(j:IDENTITY2): t <= time_out2[j]
   AND (EXISTS(k:IDENTITY1): t = time_out1[k]
   OR EXISTS(l:IDENTITY2): t = time_out2[l]);

clock: MODULE =
  BEGIN
    INPUT time_out1: TIMEOUT_ARRAY1
    INPUT time_out2: TIMEOUT_ARRAY2
    OUTPUT time: TIME
  INITIALIZATION
    time = 0
  TRANSITION
  [ time_elapses:
      time_can_advance(time, time_out1, time_out2) -->
        time' IN { t: TIME |
            is_next_event(t, time_out1, time_out2) }
  ]
  END;
```

The complete system `ttcan_faultfree` is formed by taking the asynchronous composition of the modules for potential time masters, and general nodes, and the clock module.

5.2 Failure Modeling

To model fault-tolerant startup only node failure is taken into account. It is enough to consider only the failure of the active time master, as only the failure of the active time master affects the startup procedure. In TTCAN the failures are of failstop kind, that means that when a node becomes faulty, it is no longer capable of participating in the execution of the algorithm. When a potential time master which is currently not the active time master, or a general node fails it would not affect the startup procedure.

The state diagram of a TTCAN node for fault-tolerant startup is shown in Figure 3. All the transitions except the transition from *sync* state to *wait* state are possible for a potential time master. In this model we add a new state called *faulty* to denote the state of a faulty potential time master. That the active time master is faulty is detected by all the potential time masters and all the general nodes by detecting a missing reference message. The missing reference message is captured by setting its identifier to 0. As 0 does not correspond to the identifier of any potential time master, a 0 in the reference message identifier conveys the potential time masters and the general nodes the fact that the startup algorithm has to be performed again.

At the very beginning of the startup process when there is an active time master and some potential time masters are in the *wait* state, but no node is in *arb_in_next_basic_cycle* or *sync* state, one of the potential time masters in the *wait* state should become the active time master in case of failure of the active time master. As the nodes in the *wait* state are not synchronized,

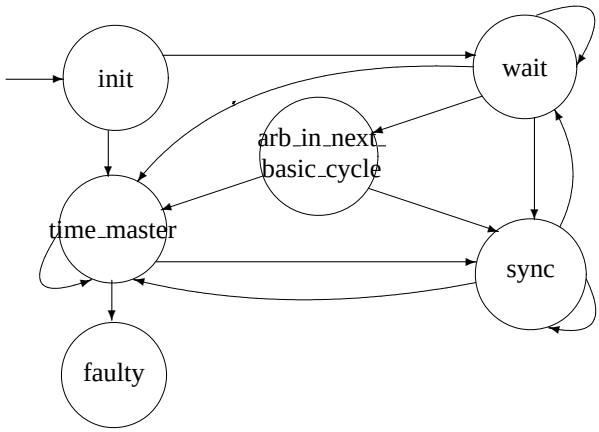

Figure 3. State diagram of a node for fault-tolerant startup

they cannot participate in the arbitration process. Thereafter moving to the *wait* state, a potential time master waits for a basic cycle to receive a reference message. If no reference message is received during that time, it becomes the active time master by sending its own reference message. Note that when the active time master is faulty and all the potential time masters are in the *wait* state, whose timeout occurs first among all the potential time masters in the *wait* state will be the next active time master and it will send the reference message immediately. All the other potential time masters in the *wait* state will receive that reference message before their timeouts occur. If the timeouts occur for more than one potential time masters in the *wait* state at the same time, arbitration procedure resolves the contention and the potential time master with the highest priority among them becomes the active time master. At the time of the execution of the startup algorithm, if the active time master fails and there are some potential time masters in *arb_in_next_basic_cycle* state then the highest priority potential time master will be the next active time master through arbitration. If there is no nodes in *arb_in_next_basic_cycle* state but some nodes in the *sync* state, then node with the highest priority among them becomes the active time master through arbitration (as they are synchronized, they can participate in arbitration process).

To model the failure of the active time master it is required to capture the time when it becomes faulty. This should be chosen randomly. We use a variable called `random_fault` for every potential time master, which denotes the number of basic cycles after which the node will be faulty. In the initialization section this variable is set to a random natural number between 1 and 100 as follows:

```
random_fault' IN {x:NATURAL | x > 1 AND x < 100};
```

Another local variable, `fault_counter` is used for every potential time master and it is set to zero during the initialization. Whenever the active time master sends a reference message, and other potential time masters receive a reference message, their `fault_counter` values are increased by 1. When it reaches its `random_fault` value, the state of active time master is changed from *time_master* to *faulty* by the following guarded command

```
[ ] time_master_to_faulty:
   pc = time_master AND time = timeout
        AND fault_counter = random_fault -->
      pc' = faulty;
      fault_flag' = TRUE;
      ref_message_calendar' =
        set_reference_message(ref_message_calendar, 0);
      winner_id' = 0;
```

The guarded transition for a node in the wait state to *time_master* state at the failure of the current active time master is shown below:

```
[ ] wait_to_time_master:
  pc = wait AND time = timeout
  AND ref_message_calendar.master_id = 0
  AND potential_time_master_higher_priority?(i) = FALSE
  AND fault_counter /= random_fault -->
    pc' = time_master;
    ref_message_calendar' =
      set_reference_message(ref_message_calendar, i);
    timeout' = time + basic_cycle_time;
    fault_flag' = FALSE;
    fault_counter' = fault_counter + 1;
```

For the general nodes there is one more transition in fault-tolerant startup than in fault-free startup. When a general node in *sync* state can detect that active time master is faulty, it moves back to the *wait* state and waits for a reference message from the new time master. The corresponding guarded transition is shown below:

```
[ ] sync_to_wait:
  pc = sync AND time = timeout AND
               ref_message_calendar.master_id = 0 -->
    pc' = wait;
```

6 Protocol Verification

In this section we list some of the properties of the startup algorithm that we verify using SAL. In SAL, G denotes "always" and F denotes "eventually". All the verification experiments have been carried out on a Dell PC with a Pentium 4 CPU (2.6 GHz) and 2 GB RAM.

6.1 Liveness Properties

We have model-checked three liveness properties for fault-free startup model and another four liveness properties for the fault-tolerant startup model.

Liveness1. The highest priority potential time master will eventually become the active time master.

```
liveness1: THEOREM ttcan_faultfree |-
  F(EXISTS (i: IDENTITY1): pc_array1[i] = time_master
                                    AND i = 1);
```

Liveness2. All the potential time masters which will not be the active time master will eventually move to the *sync* state.

```
liveness2: THEOREM ttcan_faultfree |-
  F(FORALL (i: IDENTITY1): pc_array1[i]/=time_master =>
                           pc_array1[i] = sync);
```

Liveness3. All the general nodes will eventually go to the *sync* state.

```
liveness3:THEOREM ttcan_faultfree |-
  F(FORALL (i:IDENTITY2): pc_array2[i] = sync);
```

Liveness properties for the model with fault are as follows:

Fault-tolerant_liveness1. If the active time master becomes faulty, eventually it goes to the *faulty* state.

```
fault_tolerant_liveness1: THEOREM ttcan_fault |-
  G(EXISTS (i: IDENTITY1): pc_array1[i] = time_master
  AND fault_flag = TRUE => F(pc_array1[i] = faulty));
```

Fault-tolerant_liveness2. Once the active time master becomes faulty, one among all the other potential time masters will eventually be the active time master.

```
fault_tolerant_liveness2: THEOREM ttcan_fault |-
  G(EXISTS (i: IDENTITY1): pc_array1[i] = time_master
  AND fault_flag = TRUE=> F( EXISTS(j: IDENTITY1):
  j /= i AND pc_array1[j] = time_master));
```

Fault-tolerant_liveness3. Once the active time master becomes faulty, all the potential time masters which will not be the active time master will eventually go to the *sync* state.

```
fault_tolerant_liveness3: THEOREM ttcan_fault |-
  G(EXISTS (i: IDENTITY1): pc_array1[i] = time_master
  AND fault_flag = TRUE => F( FORALL(j: IDENTITY1):
  j/= i AND pc_array1[j]/= time_master =>
                           pc_array1[j] = sync));
```

Fault-tolerant_liveness4. Once the active time master becomes faulty, all The general nodes will eventually go to the *sync* state.

```
fault_tolerant_liveness4: THEOREM ttcan_fault |-
  G(EXISTS (i: IDENTITY1): pc_array1[i] = time_master
  AND fault_flag = TRUE => F(FORALL(j: IDENTITY2):
                           pc_array2[j] = sync));
```

# PTMs	Fault-free Startup	Fault-tolerant Startup
2	39.448	67.572
3	61.152	103.886
4	86.777	143.699
5	116.919	184.590
6	144.840	225.494
7	176.965	285.917
8	206.433	328.267
9	245.558	369.104
10	289.496	442.715

Table 1. Time required to verify liveness1 property

SAL's infinite bounded model checker (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 upto depth 40 considering upto 10 potential time masters. In each case the number of general nodes have been kept equal to the number of potential time masters. sal-inf-bmc takes close time to verify each liveness property for a particular number of nodes. Table 1 presents the runtime in seconds for verifying the property *liveness1* for fault-free and fault-tolerant startup.

6.2 Safety Property

The goal of the stratup protocol is to ensure that all the potential time masters and general nodes that are in the *sync* state are synchronized with the active time master. This property can be expressed in SAL by the following LTL formula with linear arithmetic constraints:

```
safety: THEOREM ttcan_faultfree |-
 G(FORALL(i,j:IDENTITY1):FORALL(k,l: IDENTITY2):
 (pc_array1[i]=time_master OR pc_array1[i]=sync)
 AND (pc_array1[j]=time_master OR pc_array1[j]=sync)
 AND pc_array2[k] = sync AND pc_array2[l] = sync
 AND time < time_out1[i] AND time < time_out1[j]
 AND time < time_out2[k] AND time < time_out2[l] =>
 time_out1[i] = time_out1[j]
 AND time_out2[k] = time_out2[l]
 AND time_out1[i] = time_out2[k]);
```

`pc_array1` and `pc_array2` are the arrays which represent the states of Potential time masters and general nodes respectively. `time_out1` and `time_out2` are the arrays holding the timeout values for potential time masters and general nodes respectively.

This LTL formula represents the fact that on the completion of the startup algorithm, the basic cycles of all the potential time masters (including the active time master) and the general nodes will start at the same point of time.

# PTMs	*Fault Free Startup*			
	Lemma		Theorem	
	depth	time	depth	time
2	6	22.039	4	11.820
3	8	224.975	6	192.882

Table 2. Depth and time for verification of safety property for fault-free startup model by the method of induction using sal-inf-bmc

# PTMs	*Fault Tolerant Startup*			
	Lemma		Theorem	
	depth	time	depth	time
2	6	24.32	4	14.508
3	8	224.009	6	199.132

Table 3. Depth and time for verification of safety property for fault-tolerant startup model by the method of induction using sal-inf-bmc

6.2.1 Proof by Induction.

A direct method to prove the above property is by using k-induction method which is supported by sal-inf-bmc. But trying that for upto depth 10 shows that the property is not inductive. Increasing k is not of much use as the safety property is not inductive for any k. For any $k > 0$, one can find a sequence of transitions $\sigma_0 \rightarrow \ldots \rightarrow \sigma_k$ such that the safety property holds in the states $\sigma_0, ...\sigma_{k-1}$ but not in σ_k. As a consequence, the inductive step in the proof k-induction fails. The trajectory mentioned above, for a fixed k, can be found by invoking sal-inf-bmc with the -ice option. By analyzing the counterexample, it is revealed that the following lemma is required to prove the safety property.

```
safety_aux_0: LEMMA ttcan_faultfree |-
 G(FORALL(i:IDENTITY1):FORALL(j: IDENTITY2):
 time < time_out1[i] AND time < time_out2[j]);
```

Using this lemma, safety property can be proved for a model containing upto 3 potential time masters (number of general nodes have been kept equal to the number of potential time masters). Safety property for the fault-tolerant startup model can also be verified by using the same auxiliary lemma for 3 potential time masters.

In Table 2 and Table 3 we present the minimum depth required and the runtime of sal-inf-bmc in seconds to prove correctness of the safety property and the associated lemma for the fault-free model and the model considering fault respectively.

6.2.2 Proof by Abstraction.

Although the previous method is straightforward, but the method cannot prove the safety property for more than 3 potential time masters. Induction depth required to prove the safety property increases with the number of potential time masters. To make the proof scalable, the property should be proved at depth 1. To do that strengthening the invariant seems to be a good option. For this purpose, we use abstraction methodology based on verification diagram proposed by Rushby [15] and used by Dutertre and Sorea in [3]. Given a transition system $\mathcal{M} =< S, I, \rightarrow >$, this method amounts to constructing an abstraction of $\mathcal{M}$ based on n state predicates $A_1(\sigma)$, . . . , $A_n(\sigma)$. The abstraction is a transition system $\mathcal{M}_0 =< S_0, I_0, \rightarrow >$ with S_0 being the set of states and I_0 being the set of initial states. The abstract states are in a one-to-one correspondence with the n predicates. The abstract system aims to capture the fact that if A_i is true in the current state, then the next state will satisfy A_{j_1} or ... or A_{j_k}. It also states that some of the predicates $A_1, \ldots, A_n$ are true in all the initial states of $\mathcal{M}$. The correctness of the abstraction implies that $(A_1 \vee \cdots \vee A_n)$ is an inductive variant of $\mathcal{M}$.

Execution of the fault-free startup protocol can be decomposed into successive phases. Figure 4 shows the abstract system derived from A1 to A7. In the first phase A1, all the nodes are in *init* state or *wait* state. Phase A2 starts when one potential time master broadcasts a reference message and moves to *time_master* state. Collision may occur in state A2 as more than one potential time masters may broadcast reference messages almost at the same time, and this collision is resolved by bitwise arbitration. In phase A3, all the potential time masters which are not active time masters receive the reference message. In phase A4, one potential time master is in *time_master* state, and all the other potential time masters are in *init*, *wait* or *sync* state. Phase A5 starts when the active time master transmits the second reference message. If there is no node in the *wait* state or there are some nodes in the *wait* state with priority lower than that of the active time master, but without any potential time master with higher priority, then the system goes back to phase A3. The potential time masters in the *wait* state move to *sync* state. But if there is at least one potential time master with priority higher than current active time master, then the system

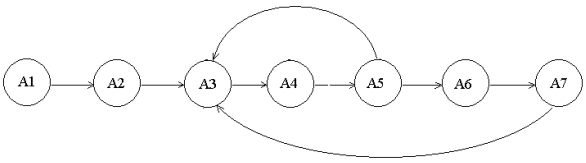

Figure 4. Verification Diagram for faultfree TTCAN startup

moves to phase A6. In phase A6, at least one potential time master is in *arb_in_next_basic_cycle* state. Phase A7 starts when the active time master and the potential time masters in the *arb_in_next_basic_cycle* state start the arbitration process by sending there own reference messages. The highest priority potential time master in the *arb_in_next_basic_cycle* state moves to *time_master* state, and the previous active time master and other potential time masters in *arb_in_next_basic_cycle* state move to *sync* state. The system moves to phase A3 again.

The abstraction predicates $A_i's$ have been defined as boolean state variables. For example A1 is defined as follows:

```
A1 =  bus_traffic = FALSE
 AND  winner_id = 0
 AND  FORALL (i: IDENTITY1): pc_array1[i] = init
 AND  FORALL (i: IDENTITY2): pc_array2[i] = init
                          OR pc_array2[i] = wait
 AND  FORALL (i: IDENTITY1): pc_array1[i] = init =>
 ref_message_calendar.not_received_flag1[i] = FALSE
 AND  FORALL (i: IDENTITY2): pc_array2[i] = init
                          OR pc_array2[i] = wait =>
 ref_message_calendar.not_received_flag2[i] = FALSE
 AND  ref_message_calendar = EMPTY_CALENDAR;
```

To prove that the abstraction is correct we construct a `monitor` module whose input variables are A1 to A7. The `monitor` is defined in such a way that it moves from one correct abstract state to another correct one if the system shows correct behavior, otherwise it moves to bad state.

The system is defined as the synchronous composition of the modules `ttcan_faultfree`, `abstractor` and `monitor`. The `abstractor` is correct if the error state is never reachable. We show this by proving the invariant property $G(state \neq bad)$. To prove this invariant lemmas are required, proving that in any correct abstract state the system shows behavior as defined in the abstract module. The correctness of the abstraction is demonstrated by proving the invariant by sal-inf-bmc using k-induction at depth 1. Finally, we prove the safety property by using the previous invariant as a lemma at depth 1.

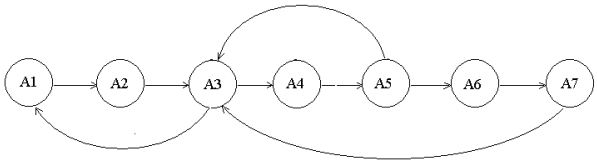

Figure 5. Verification Diagram for fault-tolerant TTCAN startup

Figure 5 shows the verification diagram for fault-tolerant startup protocol. In fault-tolerant startup, the time master can become faulty at any moment. We assume that it cannot be faulty when it is sending reference message. So, in this case time master can be faulty in state A3 or A6. If the active time master becomes faulty in phase A3, the system moves to phase A1, and one of the potential time masters whose timeout occurs first, comes to *time_master* state by sending a reference message and the system comes to phase A2. In phase A6, if the active time master becomes faulty, then that will not change the verification diagram of fault-tolerant startup from that of fault-free startup. Because in phase A6, at least one potential time master is in *arb_in_next_basic_cycle* state, and one of them will be the next active time master at the beginning of the next basic cycle.

Table 4 and Table 5 present the runtime in seconds to prove the auxiliary lemmas, the abstraction lemma, and the safety property for both fault-free and fault-tolerant startup. For fault-free startup verification experiments have been carried out for models considering upto 10 potential time masters. For fault-tolerant startup verification has not been possible for models with more than 9 nodes due to memory limitation.

# PTMs	Lemma	abstraction	safety	Total
2	24.06	3.26	3.57	30.89
3	38.42	4.98	7.54	50.94
4	74.32	9.52	16.91	100.76
5	141.44	17.99	35.23	194.66
6	308.55	38.34	71.35	418.24
7	656.55	82.37	142.54	881.46
8	1283.56	160.82	260.19	1704.58
9	2493.29	314.43	469.16	3276.88
10	3920.13	561.60	821.13	5302.86

Table 4. Time required in seconds to verify safety property for fault-free startup by abstraction method

# PTMs	Lemma	abstraction	safety	Total
2	34.61	4.163	4.87	43.64
3	64.64	7.92	10.26	82.81
4	127.54	15.79	23.40	166.72
5	307.92	38.85	56.45	403.22
6	764.08	95.96	122.19	982.24
7	1713.31	214.41	263.96	2191.68
8	3465.06	431.54	520.80	4417.40
9	6660.17	829.16	981.07	8470.39

Table 5. Time required in seconds to verify safety property for fault-tolerant startup by abstraction method

7 Discussion

In this work we have developed formal models of fault-free and fault-tolerant TTCAN startup protocol. To our knowledge, this is the first work on formal modeling of TTCAN startup protocol and its verification. Towards that we have proposed a modeling framework called Synchronous Calendar which is needed to model synchronous communication between TTCAN nodes, where no transmission delay is assumed.

Synchronous Calendar may be useful to model different kinds of protocols where link failure has to be taken into account. It can be also applied to model the drop of a message in the channel. In this case channels are to be modeled separately. In the channel process the flag corresponding to the node whose message has to be dropped or whose channel to the sender is faulty can be made false. It is also possible to model unbounded message delay by suitably adapting the synchronous calendar data structure. Message delay over a particular limit is generally considered as message lost, and this fact can be handled by properly defining the timeouts.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] 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 Fifth NASA Langley Formal Methods Workshop, NASA Langley Research Center*, pages 187–196, 2000.

[3] 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.

[4] B. Dutertre and M. Sorea. Timed systems in sal. Technical report, Computer Science Laboratory, SRI Internationalhh, 2004.

[5] T. Führer and A. Schedl. The steer-by-wire prototype implementation: Realizing time triggered system design, fail silence behavior and active replication with fault-tolerance support. In *Proceedings of SAE*, 1999.

[6] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time triggered communication on CAN (Time-Triggered CAN-TTCAN). 2000. Available at http://www.semiconductors.bosch.de/pdf/CiA 2000Paper_1.pdf.

[7] H. Hartwich, B. Müller, T. Führer, and R. Hugel. Timing in the TTCAN network. Available at http://www.can-cia.org/can/ttcan/hartwich2.pdf.

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

[9] ISO11898-Part1. Road vehicles - interchange of digital information - part 1: Controller Area Network (CAN) for high-speed communication. 1993.

[10] H. Kobayashi. *Modeling and Simulation: An Introduction to System Performance Evaluation Methodology*. Addison-Wesley, New York, 1981.

[11] H. Kopetz. The time-triggered model of computation. In *Real Time Systems Symposium*. IEEE Computer Society, 1998.

[12] L. M. Moura, S. Owre, H. Rueß, J. M. Rushby, N. Shankar, M. Sorea, and A. Tiwari. Sal 2. In *CAV*, volume 3114 of *LNCS*, pages 496–500. Springer, 2004.

[13] L. Pike and S. Johnson. The formal verification of a reintegration protocol. In *Proceedings of the 5th ACM international conference on Embedded software(EMSOFT 2005)*, pages 286–289, 2005.

[14] A. Pritsker and C. Pegden. *Introduction to Simulation and SLAM*. Wiley, New York, 1979.

[15] J. Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In *Proceedings of Computer-Aided Verification*, volume 1855 of *LNCS*, pages 508–520. Springer-Verlag, 2000.

[16] W. Steiner. Model-checking studies of the flexray startup algorithm. Research Report 57/2005, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2005.

[17] 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.

[18] N. Weininger and D. Cofer. Modeling the ascb-d synchronization algorithm with spin: A case study. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 93–112. Springer-Verlag, 2000.