An UPPAAL Model for Formal Verification of Master/Slave Clock Synchronization over the Controller Area Network

Guillermo Rodríguez-Navas, Julián Proenza
SRV, Dept. de Matemàtiques i Informàtica
Universitat de les Illes Balears, Spain
guillermo.rodriguez-navas@uib.es
julian.proenza@uib.es

Hans Hansson
Malardalen Real-Time Research Centre
Dept. of Computer Science and Electronics
Mälardalen University, Sweden
hans.hansson@mdh.se

Abstract

Many distributed applications require a clock synchronization service. We have previously proposed a clock synchronization service for the Controller Area Network (CAN), which we have claimed to provide highly synchronized clocks even in the occurrence of faults in the system. In this paper we substantiate this claim by providing a formal model and verification of our fault tolerant clock synchronization mechanism. We base our modeling and verification on timed automata theory as implemented by the model checking tool UPPAAL. In the modeling we introduce a novel technique for modeling drifting clocks. The verification shows that a precision in the order of 2 μs is guaranteed despite node's faults as well as consistent channel faults. It also shows that inconsistent channel faults may significantly worsen the achievable precision, but that this effect can be reduced by choosing a suitable resynchronization period.

1. Introduction

One reported liability of the Controller Area Network (CAN) is its lack of a clock synchronization service [1, 2]. Due to this, whenever a CAN-based distributed embedded system requires a synchronized clock, such service has to be provided at the application level, usually by means of a software-implemented clock synchronization algorithm [3, 4, 5].

We have proposed a particular solution for clock synchronization over CAN networks [6], which we claimed is able to provide a clock of high precision despite the occurrence of a wide variety of faults in the system. This paper is intended to substantiate this claim. More specifically, the aim of this work is to formally verify what precision can be achieved and to find out how certain faults affect the best achievable precision.

In our verification we use timed automata theory as implemented by the UPPAAL model checker [7]. Timed automata are very suitable for verifying real-time systems as they incorporate the notion of time. However, all required elements for our modeling are not directly supported. In particular, drifting clocks –which are essential in our modeling- are not supported. Modeling such clocks is one important contribution of this work.

The first steps towards the formal verification of our clock synchronization were described recently in [8]. In that work, the first version of our verification model was described and a few preliminary properties were verified. In this paper we present the complete formal verification of the clock synchronization service. Furthermore, as an important contribution, we provide quantitative results under diverse fault scenarios, which therefore allow us to assess the impact of certain fault hypothesis on the precision of the clock. Thus, we show how model checking can be used in order to make suitable trade-off decisions on certain parameters of the clock synchronization service.

The rest of the paper is organized as follows. In Section 2 the context of this research is discussed in order to further motivate our work. Section 3 describes the main features of our solution for clock synchronization over CAN, as required for the understanding of the work. Section 4 is devoted to describing the main features of our verification model, whereas the results of the formal verification are presented and discussed in Section 5. Finally, Section 6 summarizes the paper and gives some insight for further research.

2. Motivation

CAN is a fieldbus technology that can be considered a *de facto* standard for low-cost distributed embedded systems. It is nowadays used not only in the automotive industry, but also in many other fields such as factory automation, medical equipment and building automation, among others [9].

The enormous popularity of CAN makes it a very cost-competitive technology. Due to this, substantial efforts have been made to overcome the limitations that prevent its adoption in other application fields. In particular, there have been several proposals that aim at improving the properties of CAN from a dependability perspective, for instance by reducing the response time indetermin-

ism [4, 10, 11] or by providing tolerance to babbling idiot failures [12, 13].

Many of these mechanisms rely on the assumption that the nodes share a common time base. However, only one of them, the Time Triggered CAN (TTCAN) protocol [4], describes how this clock synchronization should be achieved, though it has not been formally verified. On the other hand, even though generic solutions for clock synchronization over CAN exist [3, 5], and can therefore be adopted, none of them has been formally verified.

This clearly contradicts what has been done with other technologies for dependable distributed embedded systems. For instance, the development team of the Time Triggered Architecture (TTA) claims that the clock synchronization service is one of their basic mechanisms [14] and thus they start the formal verification of their architecture by verifying this service [15]. Once they have verified this, they assume that a common time base exists in order to verify the rest of mechanisms.

In this paper, we use model checking to formally verify that our clock synchronization mechanism behaves as intended in assumed fault scenarios. Model checking is one of the most commonly used techniques for formal verification, which is capable of automatically determining whether an operational model (in our case a network of timed automata) satisfies properties expressed in a property language (typically some form of modal logic).

The advantages of model checking are that it explores the complete state space and that it is fully automatic (once the model and properties are specified). This is in contrast with simulation and testing, which typically only explore a small fraction of the state space, thereby only being useful for showing the presence of faults (not their absence).

3. Our solution for clock synchronization over CAN

This section is devoted to describing the main characteristics of our solution for clock synchronization over CAN. Due to space limitations, only those characteristics that are relevant for the formal verification are discussed. Further information can be found in [6].

3.1. System architecture

A CAN network is composed by a number of so-called CAN nodes, which execute a coordinated function and use a broadcast network to exchange messages. The structure of a CAN node is depicted in Figure 1. It is made up of two basic elements: a *processor*, which executes both the application software and the clock synchronization algorithm, and a *CAN controller*, which provides the processor with the CAN communication capabilities.

Note that the processor is clocked by a quartz local oscillator. This local oscillator is used to measure time, by means of a software counter which simply acts as frequency divider. This software-implemented counter is

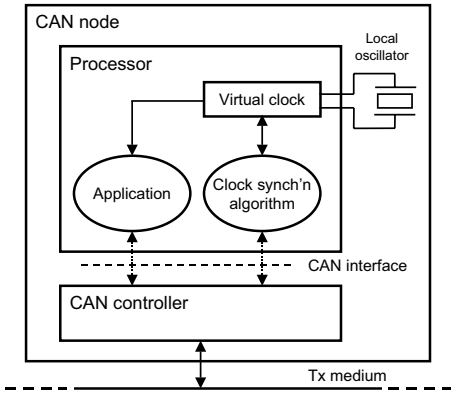

Figure 1. Structure of a CAN node executing a clock synchronization algorithm

called the *virtual clock*. We hereafter refer to the virtual clock of node i as $VC_i(t)$.

Due to fabrication imperfections, aging, and other environmental parameters, the local oscillators of a network cannot tick at exactly the same frequency. Thus, as long as no correction action is performed, virtual clocks tend to drift apart.

In order to guarantee that the virtual clocks of the various CAN nodes do not drift apart *too much*, each node executes a clock synchronization algorithm. The function of this algorithm is to compare the value of the local virtual clock with a valid reference and, if needed, to make the appropriate corrections to the virtual clock. The valid reference is obtained by exchanging synchronization messages through the CAN network.

3.2. Protocol rationale

Our clock service addresses the problem of internal clock synchronization. Thus, it is intended to guarantee the following property:

P1 (Precision): For every pair of nodes i, j

$$|VC_i(t) - VC_j(t)| \leq \Pi \quad \forall t \tag{1}$$

Our protocol adopts a master/slave approach to fulfill this goal as there is one node (the *master*) that imposes its time view to the rest of the nodes of the system (the *slaves*). The master is assumed to be synchronized to an external time source which supplies the desired accuracy.

The master's time view is spread throughout the network by means of a specific message, the *Time Message* (TM), which the master periodically broadcasts. The aim of the Time Message is twofold. On the one hand, the master uses it to indicate the resynchronization event, which is the sample point of the *Start of Frame* bit of this particular message [6]. Every slave takes a sample of its own virtual clock at this instant and keeps it in a local variable called lt_ref. On the other hand, the TM's data

field conveys the timestamp that the master took precisely at that instant, so once the TM is received, each slave extracts the master's timestamp and saves it in a local variable called rt_ref.

After that, every slave corrects its clock with respect to the master. In particular, two corrections are performed. One aims at correcting the *offset error*, which is calculated according to Equation 2, whereas the other aims at correcting the *drift error*, which is caused by the rate difference between the local virtual clock and the remote virtual clock, as shown in Equation 3.

$$\textit{offset\_error} = rt\_ref - lt\_ref \tag{2}$$

$$\textit{drift\_error} = \frac{\textit{local\_rate}}{\textit{remote\_rate}} = \frac{lt\_ref - rt\_ref_{prev}}{rt\_ref - rt\_ref_{prev}} \tag{3}$$

These corrections are not performed instantly, but they are progressively carried out [6]. This is often called *clock amortization* and is more advisable than instant corrections because it does not cause time leaps.

Both offset correction and drift correction are never perfectly performed, since there is always some residual error caused by small system latencies or by the imprecision of the arithmetical operations. However, in our system the residual error of the offset correction is small, mainly because message timestamp is implemented in hardware, and can thus be neglected. In contrast, the drift error (though being small) may have greater effect on the clock precision because it makes clocks drift apart as time goes by, and therefore it cannot be neglected. The residual drift error is denoted γ_{min} hereafter. The maximum drift error between any pair of slaves that have synchronized to the same master is $2 \times \gamma_{min}$.

3.3. Fault model

The system may suffer from three types of faults: faults in the software design, physical faults of the nodes and physical faults of the channel. Due to the simplicity of our clock synchronization protocol, faults of the software design are not included in our fault model.

Concerning physical faults of the nodes, our fault model includes arbitrary faults that can be either transient or permanent. However, an important assumption is that there is at least one non-faulty node in the system which can provide a clock of good quality, so there is always one *eligible* master.

Permanent physical faults of the channel, such as bus partition or stuck-at-dominant, are not included in the fault model. These faults can be addressed independently, for instance as proposed in [16].

Regarding transient channel faults, our fault model only includes those channel faults that lead to frame errors detectable by the error-detection mechanisms of CAN, since the probability of having a channel fault which is not detected by the CAN controllers is very low and can be neglected [17]. It is also assumed that the number of channel faults within a given time interval is bounded. In this way, it is possible to assume a bounded response time for every CAN message [18].

It is important to remark that our fault model considers transient channel faults that can lead to inconsistent frame receptions (both inconsistent omissions and inconsistent duplicates). Fault scenarios that may cause such inconsistencies have been reported in [19, 20]. Nevertheless, the probability of such faults is low enough so as to assume that only a bounded number of consecutive resynchronization rounds can be affected by these faults.

3.4. Fault tolerance aspects

The main drawback of adopting a master/slave scheme is that the master represents a single point of failure. In our proposal, this single point of failure is eliminated by means of *master replication*. Basically, a number of master replicas or *backup masters* are defined, which supervise the so-called *active* master so one of them can take over in case this master is faulty.

The instant at which each master attempts to transmit the TM is called its *release time* (T_{rlsi}). The distribution of the release times as well as the complete structure of the resynchronization round is depicted in Figure 2. Notice that the resynchronization period is R, and that each resynchronization round can be divided into two segments. TMs are exchanged only within the first half of the resynchronization round.

In order to simplify the management of the master replication, the failure semantics of the masters (either active or backup) has been restricted to *crash failure semantics*, which means that upon a physical fault the node becomes silent [6]. Thanks to this, the faulty master detection and replacement mechanism is implemented by just defining the release time of the backup masters a short time after the release time of the active master.

Even though the release time is periodical, the instant at which the TM is actually broadcast may vary depending on the network conditions. For instance, channel errors may cause frame retransmissions and therefore a significant delay. Nevertheless, since the worst case response time of the TM is bounded, TMs are exchanged only for a limited amount of time, which we call *TMdelay*.

As it was stated in Section 3.3, certain channel faults may cause inconsistent omissions or inconsistent duplicates of the TM. Inconsistent TM duplicates do not have any negative impact on the clock precision because they do not cause inconsistent resynchronizations. In contrast, inconsistent omissions of the TM can make some nodes resynchronize to different masters, and hence increase their respective clock error. These potential inconsistency scenarios and their effect on the clock parameters are explained next:

A backup master or a slave may not receive the TM sent by the active master. In such a case, the node would not correct its offset error with respect to the active mas-

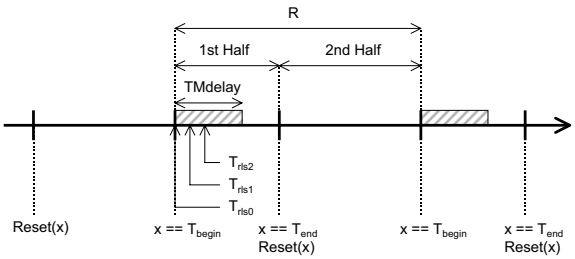

Figure 2. Structure of the resynchronization round

ter's clock. If the drift error of this node's clock after the last successful resynchronization was γ then the accumulated offset error would be at most equal to γ times the elapsed time since that resynchronization. The drift error would be neither corrected, but the effect would not be too important since clock drift tend to change smoothly and therefore, for a time short enough, the drift error can be considered as being constant.

A node (either backup master or slave) may not receive the TM sent by the active master but do receive a subsequent TM from a backup master. In such a case, these nodes correct their clock according to that backup master, and *not* according to the active master. Due to this, they actually "inherit" both the offset error and the drift error of this backup master, which might be even greater than the errors these nodes had before resynchronization. Moreover, the drift error of these nodes may increase due to the residual drift error. For instance, if γ is the drift error (w.r.t. the active master) of the backup master that transmitted the second TM, then after resynchronization the drift error of the resynchronizing nodes may be $\gamma + \gamma_{min}$.

3.5. Master's and slave's finite state machine

Figure 3 describes the behavior of the masters. This finite state machine is the basis for the verification model presented in Section 4.

Note that the master i requests transmission of its TM (TM.req!) only if T_{rlsi} is reached, and that even if the TM transmission has been requested, the master may still receive the TM from a higher priority master (TM.ind?). In such a case, it aborts the transmission of the TM previously requested (Abort.req!) and corrects its clock according to the TM just received. If the master succeeds in transmitting its own TM (TM.conf?) then it does not perform any clock correction as it considers itself the current active master.

Figure 4 shows the behavior of the slave, which is significantly simpler than the master's one: its clock is corrected after the first TM of the round is received.

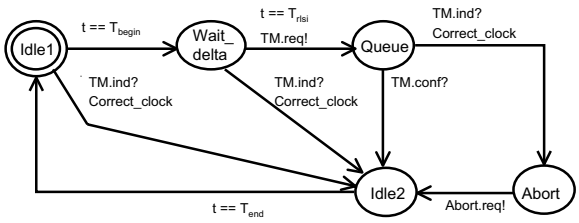

Figure 3. Master's finite state machine

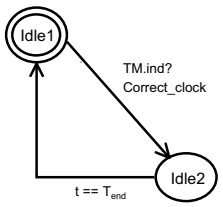

Figure 4. Slave's finite state machine

4. Verification model

As indicated in Section 2, the first step in the verification procedure consists in building a formal model of the system under verification. In our case, the model has been specified as a network of timed automata, and adheres to the syntax rules of the UPPAAL model checker.

The main challenge of this work has been to specify drifting clocks, i.e. clocks whose rate may vary throughout the verification. In the UPPAAL literature, some models can be found where clocks of different rate are used [21]. These models require every clock to have a constant rate, whereas our model allows the clock rate to change *dynamically*, as a result of the resynchronization actions that the nodes perform. To our knowledge, this work is the first one to model such clocks.

Due to its inherent complexity, and in order to facilitate the reader's comprehension, the model is presented in an incremental way. First, a global description of the model is provided and after that, every particular automaton is discussed in detail.

4.1. Model overview

Figure 5 shows the general structure of the model. It is made up of a number of automata (also called processes), which communicate through synchronization channels and/or global variables (i.e. variables that can be read and written by all of the processes).

In our model only master nodes have been included. This is a slight restriction, but we expect that it will be easy to incorporate slaves, as their finite state machine is a subset of the master's one. Furthermore, slaves are in a sense already included, since backup masters behave as a sort of slaves as long as they do not have to replace the faulty active master. Hence, the verification carried out in this work actually provides some indirect information about the precision achievable by the slaves.

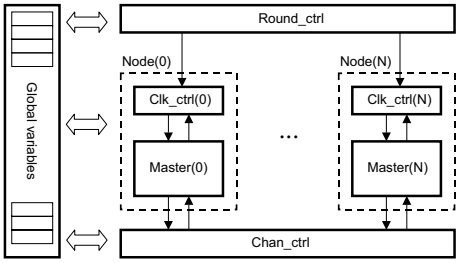

Figure 5. Overview of our UPPAAL model

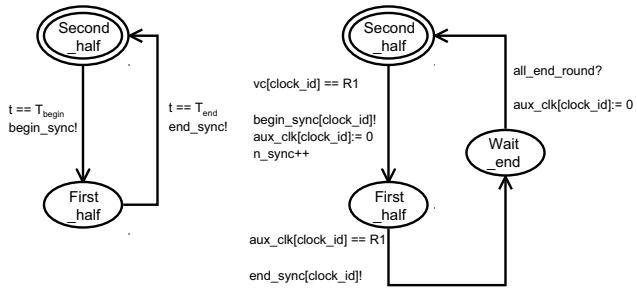

Figure 6. Simplified Clk_ctrl automata (without temporal uncertainty)

Our model includes one automaton, named `Master`, for each master in the system; this automaton models most of the clock synchronization protocol as well as some additional features such as master crash and TM omissions. Notice that there is one additional automaton, named `Clk_ctrl`, associated to every master; this automaton manages the virtual clock of each master in the way that is explained later on in this section.

Furthermore, there is an automaton, named `Chan_ctrl`, which models two relevant properties of CAN: arbitration and bounded response time. Last, the automaton labelled `Round_ctrl` is intended to check the state of certain global variables (mainly flags and counters) at the end of every resynchronization round, and modify their values when appropriate.

4.2. Modeling the master's virtual clock: the Clk_ctrl automaton

In the model, each master is provided with its own virtual clock. In particular, the global variable `vc[my_id]` represents the virtual clock of master `my_id`. Every virtual clock is managed by one `Clk_ctrl` automaton. Thanks to the local variable `clock_id`, each `Clk_ctrl` automaton knows which master it is related to.

Figure 6 shows two simplified versions of the `Clk_ctrl` automaton. The left automaton in the Figure shows the ideal behavior, which is just to signal the instants T_{begin} and T_{end} of every resynchronization round. Note that this behavior corresponds in fact to the clock x that was presented in Section 3.4, when discussing the structure of the resynchronization round (also depicted in Figure 2).

The automaton on the right side of Figure 6 is much closer to the final specification in UPPAAL language. However, and for the sake of clarity, it does not model temporal uncertainty (i.e. drifting clocks) yet. It is discussed in Section 4.4.

In `Clk_ctrl` the following values are assigned: T_{begin} = R1 and T_{end} = R, where $R1 = R/2$. This value was chosen to simplify calculations; any other value can be chosen as long as it guarantees that $T_{end} - T_{begin}$ is greater than $TMdelay$ (the maximum duration of the TM exchange at the beginning of the round).

Note that the `Clk_ctrl` automaton uses an additional clock, along with `vc[clock_id]`. This clock is called `aux_clk[clock_id]` and is used to measure T_{end}. The clock `aux_clock[clock_id]` is restarted as soon as `vc[clock_id]` reaches T_{begin} so it actually measures the time elapsed since the beginning of the first half of the resynchronization round (i.e. `vc[clock_id]` $- T_{begin}$).

There is an additional location, called `Wait_end`, where each `Clk_ctrl` automaton waits until every other `Clk_ctrl` automaton has reached the end of the resynchronization round. This is signaled through the channel `all_end_round`. Whenever this happens, `aux_clk[clock_id]` is restarted again, yet with the only purpose of reducing the state space.

4.3. Modeling the master's behavior: the Master automaton

Figure 7 shows the UPPAAL automaton that models the master behavior. Even though the automaton may seem rather puzzling, it can be understood easily if it is interpreted on the light of what was explained in Section 3 about the master's finite state machine.

In fact, it is easy to see that every location depicted in Figure 3 does appear in the `Master` automaton. Therefore, in order to explain the automaton we depart from these known locations and use the transitions of Figure 3 as guidelines. In this way, the automaton is introduced gradually. Nevertheless, and due to space limitations, only those features related to modeling clocks are discussed. The rest of features are described in [22, 8].

Condition (t == T_{begin}) and condition (t == T_{end})

As already explained, each `clk_ctrl` automaton signals the beginning and the end of every resynchronization round through the broadcast channels `begin_sync[clock_id]` and `end_sync[clock_id]`, respectively.

The `end_sync[clock_id]` channel is moreover used by the masters in order to correct their error with respect to the time reference. This is explained in Section 4.4, when discussing clock correction.

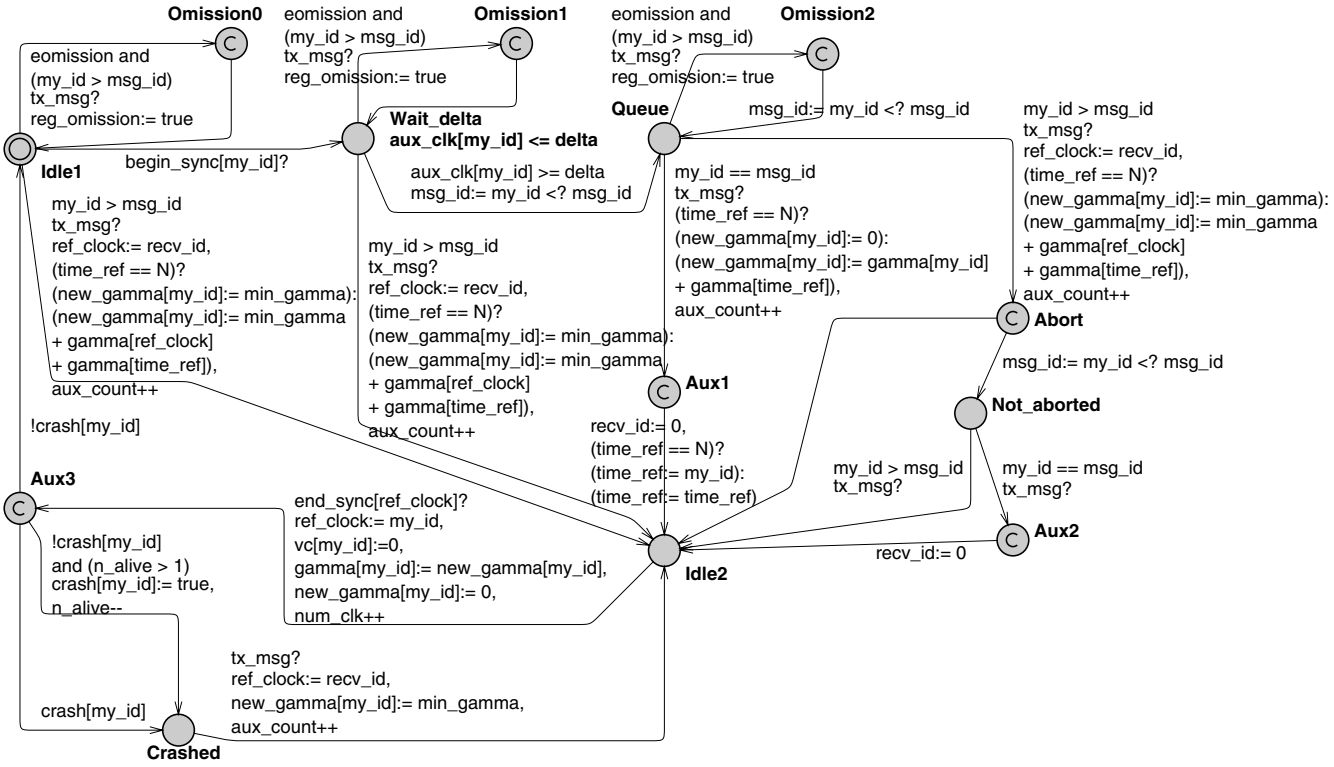

Figure 7. Master automaton (UPPAAL syntax)

Condition (t == T_{rlsi}**)**

As indicated in Section 3, every master i has a different release time, which is calculated as follows: $T_{rlsi} = T_{begin} + \Delta_i$. Thanks to the channel `begin_sync[my_id]` every master knows the beginning of the resynchronization round. As soon as the beginning of the Resynchronization round is reached, the master steps into location `Wait_delta`. In this location, the clock `aux_clk[my_id]` –which was restarted by the corresponding `Clk_ctrl` automation– is used to measure the interval Δ_i. This is modeled in UPPAAL with the time invariant `aux_clk[my_id]` $\leq$ `delta` together with the guard condition `aux_clk[my_id]` $\geq$ `delta` (see location `Wait_delta` in Figure 7).

TM request / TM confirm / TM indication

In order to model the transmission and reception of the TM, the `Master` automaton interacts with the `Chan_ctrl` automaton (depicted in Figure 8). This automaton uses the clock x to restrict the maximum transmission delay of each TM, which is in the range $[Ctm, WCRT]$, being Ctm the transmission time of one TM.

Whenever the transmission of a TM takes place, this is signaled by the `Chan_ctrl` automaton to the masters through the broadcast channel `tx_msg`. However, each `Master` may ignore this indication and remain in the same location (after stepping into one of the committed locations `Omissioni`). In this manner, inconsistent omissions of the TM are modeled. The automaton `Round_ctrl`, which is shown in Figure 9, is intended to enable/disable the possibility of TM inconsistencies by means of the global variable $eomission$.

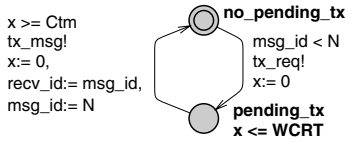

Figure 8. Chan_ctrl automaton (UPPAAL syntax)

TM abort

This action does not require any special mechanism so it is modeled as a simple transition to location `Idle2`. Nevertheless, since the abort operation may not be performed on time, there is one unconditional transition from location `Abort` to location `Not_aborted` which models such situation.

4.4. Clock correction and temporal uncertainty

In Section 3, it was mentioned that clock correction can actually be divided into two actions: offset correc-

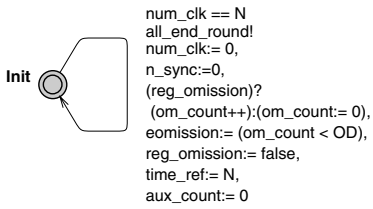

Figure 9. Round_ctrl automaton (UPPAAL syntax)

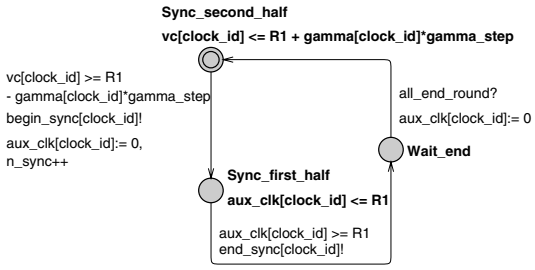

Figure 10. Clk_ctrl automaton with temporal uncertainty (UPPAAL syntax)

tion and drift correction. We model offset correction by forcing those masters that had synchronized to a given master i to restart their virtual clock simultaneously with such master. This is done in the transition from location `Idle2` to `Aux3`. Note that this transition is fired when the master receives an indication through the broadcast channel `end_sync[ref_clock]`. What this condition really means is that the virtual clock of master `ref_clock` (which is the master from which this master received the TM) has reached T_end.

To model drift correction we use an array of integers `gamma[]` that keeps the drift error between every master and the active master; so `gamma[i]` $= 0$ if master i is the active master whereas `gamma[i]` > 0 for the other masters. The drift with respect to the active master is recalculated in every resynchronization round because it may change as a consequence of the clock correction. This value is always an integer multiple of γ_0 (constant `gamma_min` in the model).

As described in Section 3, the masters correct their clocks in the transition from location `Idle1` and `Wait_delta` to `Idle2` and from location `Queue` to `Abort`. These transitions correspond to the first reception of a higher priority TM, so at this point the id of the current active master is known and kept in the variable `time_ref`. The drift error is calculated as follows:

If master i is the current active master:

```
gamma[i]:= 0
```

If master i synchronized to the current active master:

```
gamma[i]:= gamma_min
```

If master i synchronized to a master *ref_clock* that is not the current active master:

```
gamma[i]:= gamma_min
 + gamma[ref_clock] + gamma[time_ref]
```

The variable `gamma[i]` is used in our model to model the temporal uncertainty. This is illustrated in Figure 10. Note that in this automaton, the first half of the resynchronization round may start within a time interval of length $2\times$ `gamma[clock_id]` $\times$ `gamma_step`, and centered in R1. Where `gamma_step` is the imprecision that a drift error of γ_0 would cause after R t.u. (the resynchronization period).

5. Formal verification

The main property we want to verify is the *Precision* property. This property, as mentioned in Section 3, states that for any pair of clocks, their difference is never greater than a given value Π (the precision).

In this section, we describe how this property is specified and verified in our model. We also show how we take advantage of our formal model to investigate to what extent certain parameters affect the precision of the clock synchronization service.

5.1. Verification procedure: the Begin_observer automaton

In order to determine the precision between two clocks we first have to identify the instant at which they exhibit their maximum difference, and then we have to measure the clock difference at that particular instant.

It can be shown that the instant of maximum clock difference is right before the reception of the active master's TM, since from that moment on the clocks start converging to the value indicated by the TM. Therefore, we should measure the clock difference at that instant.

From what is explained in Section 3.4 about the structure of the resynchronization round, it can be observed that the TM is always received at certain instant within the interval $[T_{begin} + Ctm, T_{begin} + \mathit{TMdelay}]$. This property allows us to determine the maximum error between two clocks easily. First, we measure the difference between the two clocks at the instant T_{begin} and, second, we upper bound the extra error these two clocks may accumulate until the TM is received.

Particularly, the error that Master i and Master j may accumulate after *TMdelay* is upper bounded by N_{delay_ij}, as calculated in Equation 4; where γ_{ij} represents the relative drift error between Master i and Master j, and is therefore calculated as the sum of their drift errors with respect to the time reference (`gamma[i] + gamma[j]`, in the UPPAAL model).

$$N_{delay\_ij} = \mathit{TMdelay} \times \gamma_{ij} \quad (4)$$

A new automaton has been specified, which applies this reasoning in order to check the achievable precision. This automaton, depicted in Figure 11, is called `Begin_observer`. It acts as an external observer of the system: it periodically compares the clocks of the various masters and determines whether they may exceed the desired precision or not. This comparison is performed once per round.

In this automaton, the location `Initial` is left as soon as all of the masters have started the resynchronization round (condition `n_sync == N`). This means that the expression `(aux_clk[i] - aux_clk[j])` keeps the offset of Master i and Master j at T_{begin}. Additionally, in the transition from the committed location `Begin_sync` to the location `Wait_End`, the value of each N_{delay_ij} is recalculated.

Once in location `Wait_End`, this information is used by the automaton in order to check whether the clock difference between any pair of non-faulty masters exceeds the desired precision, which is kept in the constant `MAX_PI`. Note that if the difference (in fact, the absolute value of the difference) between two clocks may exceed this value then the transition to location `Failure` may be fired.

This makes the formal verification turn into solving a reachability problem: if the location `Failure` may be reached then it means that the clock error may exceed the desired precision. This property is specified in LTL as:

```
A[] not Begin_observer.Failure
```

If this property is satisfied by the model it means that the clocks are always within the desired precision (`MAX_PI`). Whenever this property is not satisfied, it means that the clocks may drift apart so much as to exceed the desired precision. The trace obtained can thus be analyzed in order to understand what causes the error increment.

5.2. Verification scenarios

Our model allows modification of the following parameters:

- Number of masters (N).
- Resynchronization period (R).
- Release time of the masters (Δ_i).
- Residual error after clock correction (γ_0). This parameter gives a measure of the stability of the local oscillator, and it also takes into account the error accumulated in the arithmetical operations of the clock correction procedure. This parameter is intimately related to the constant `gamma_step`, since `gamma_step`$= \gamma_0 \times$ R.
- Network load. The variable `WCRT` in `Chan_ctrl` models the delay caused by channel errors that lead to frame retransmission as well as the delay caused by higher priority traffic on the bus.
- Master faults. It is possible to set the maximum number of masters that may crash during the verification. No assumption is made on the order in which the masters crash.
- Data consistency. Inconsistent message omissions can be enabled/disabled. When enabled, it is possible to bound the maximum number of consecutive resynchronization rounds that can suffer from inconsistencies. We refer to it as the *Omission Degree*. No assumption is made on the spatial distribution of the inconsistent omissions so that every possible combination of message inconsistency is checked.

The scenarios we verify include the following situations: 1) Fault-free scenario; 2) Only master faults scenario; 3) Only channel faults scenario, assuming *data consistency* and without assuming *data consistency*; and 4) Master faults and channel faults scenario, assuming *data consistency* and without assuming *data consistency*.

5.3. Verification results

In order to check the correctness of the model, some preliminary properties were also checked. These properties were intended to verify that the system assumptions hold throughout the verification. For instance, it was checked that the maximum number of allowed master crashes was never exceeded or that the omission degree value was neither exceeded. Due to space limitations, these properties are not discussed in this paper. As an example, we only show the property that is verified in order to check that there is always at least one non-faulty master in the system:

```
A[] not (crash[0] and crash[1]
      and crash[2] and crash[3])
```

Concerning the precision guaranteed by the clock synchronization service, Table 1 shows the precision that was verified under diverse fault assumptions. These results were obtained with the following parameters: N= 4 masters, R= 1s, Δ_0= 0, Δ_1= 1 ms, Δ_2= 2 ms, Δ_3= 2 ms. Regarding the network load, it was assumed that no other messages where sent on the bus, so WCRT= 1.04 ms was used in those scenarios without channel faults whereas WCRT= 6 ms was used in those scenarios with channel faults. Anyway, the results show that the network load is not very relevant for the clock precision.

The first cell in Table 1 shows the precision guaranteed in the fault-free scenario. This precision equals to 2 μs. The first row of Table 1 corresponds to the scenarios in which only master's faults were assumed. Note that the number of faulty master does not affect significantly the precision guaranteed. Unfortunately, the limited temporal granularity of our model makes it impossible for us to measure the difference between having one or more faulty masters. Nevertheless, in some simplified verifications we have reported that this difference is in the order of 10 ns.

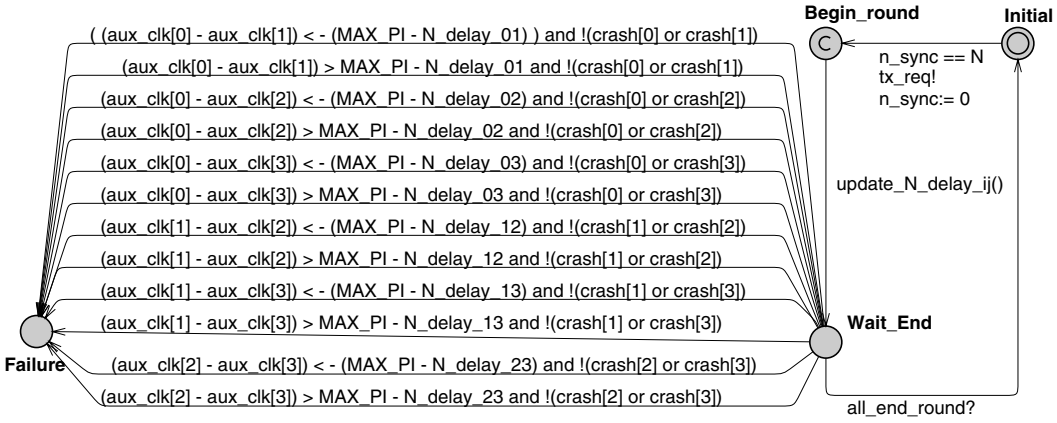

Figure 11. Automaton Begin_observer (UPPAAL syntax)

Table 1. Fault assumptions and precision guaranteed (in μs) with R = 1 sec

# Channel faults	# Faulty masters			
	0	1	2	3
No faults	2	2.1	2.1	2.1
OD = 0	2.1	2.1	2.1	2.1
OD = 1	6.1	6.1	6.1	6.1
OD = 2	10.1	12.1	12.1	12.1
OD = 3	14.1	16.1	16.1	16.1

Table 2. Fault assumptions and precision guaranteed (in μs) with R = 0.5 sec

# Channel faults	# Faulty masters			
	0	1	2	3
OD = 0	1.1	1.1	1.1	1.1
OD = 1	3.1	3.1	3.1	3.1

The first column of Table 1 corresponds to the scenarios in which only channel's faults were assumed. The parameter OD stands for Omission Degree and refers to the number of resynchronization rounds that can suffer from TM inconsistent omissions. Thus, OD= 0 indicates that no inconsistent omissions can occur, which is a common assumption in other clock synchronization protocols for CAN.

The rest of cells in Table 1 correspond to the scenarios where a combination of node's and channel's faults is assumed. In particular, the right bottom cell corresponds to the most severe fault scenario.

Table 2 shows some of the results obtained when the resynchronization period is reduced to 0.5 s. These results prove the intuition that the negative effect of the inconsistencies may be reduced by synchronizing more frequently.

5.4. Discussion

The results obtained show that certain failures have greater impact on the precision. Particularly, it is seen that inconsistent message omissions affect more negatively than master crashes. It is curious that even though master crashes are usually addressed by current solutions for clock synchronization, very little attention is paid to the fact that message inconsistencies may occur [4].

It may be argued that message inconsistencies are very unlikely in CAN networks, but certain authors claim that the probability is such that it should be taken into account when designing fault-tolerant systems for dependable applications [19, 20]. Moreover, it has been reported that the probability of message inconsistencies in TTCAN increases dramatically when compared to the so-called natural CAN [23].

6. Conclusion

In this work we have used the UPPAAL model checker in order to verify that our solution for clock synchronization over CAN achieves the desired precision even in the presence of various node's and channel's faults. The formal verification also showed that inconsistent channel faults are a severe threat to the clock precision, but that their negative impact can be reduced by choosing a suitable resynchronization period.

Thus, model checking turned out to be a useful tool not only for checking the correctness of the system, but also as a tool to assist the system designer. Even though building the verification model is not a trivial task, once it is finished then it is possible to carry out many verifications under diverse circumstances. This helps the system designer to better understand the trade-offs of the system, and hence make more appropriate decisions.

In order to model our system, a novel technique for modeling drifting clocks was developed. To our knowledge, this is the first work that models such clocks with timed automata.

During the formal verification, we had to be careful to keep the memory required for the verification bounded. This state-space problem is a well-known problem of model checking. We found out that time granularity was

the main difficulty since finer granularity could only be achieved at the cost of significantly increasing the state space. By increasing the time granularity, the number of states to be traversed does not change, but the amount of memory required to keep every state changes significantly. This makes the state space blow up, and increases dramatically the verification time.

In the future, we would like to link our verification model to some kind of tool for dependability evaluation so we could assess the likeliness of the most severe scenarios. Furthermore, we have realized that this work can be included within the framework of a more general problem: the formal verification of *hybrid systems* [21]. Thus, we would like to investigate whether the novel techniques we have applied in our model can be also applied in order to verify other kind of hybrid systems, such as distributed embedded systems based on sensor/actuator networks.

Acknowledgement

This work has been partially supported by the Spanish CICYT grant DPI2005-09001-C03-02, which is partially funded by the European Union FEDER programme.

References

[1] ISO, "ISO11898. Road vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication", 1993.

[2] M. Törngren, "A perspective to the Design of Distributed Real-time Control Applications based on CAN", *Proceedings of 2nd International CAN Conference, London-Heathrow, United Kingdom*, 1995.

[3] L. Rodrigues, M. Guimaraes, and J. Rufino, "Fault-tolerant Clock Synchronization in CAN", *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain*, 1998.

[4] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, M. Walther, and R. B. GmbH, "Time Triggered Communication on CAN", *Proceedings of the 7th Int. CAN Conference, Amsterdam, The Netherlands*, 2000.

[5] D. Lee and G. Allan, "Fault-tolerant Clock Synchronisation with Microsecond-precision for CAN Networked Systems", *Proceedings of the 9th International CAN Conference, Munich, Germany*, 2003.

[6] G. Rodríguez-Navas, J. Bosch, and J. Proenza, "Hardware Design of a High-precision and Fault-tolerant Clock Subsystem for CAN Networks", *Proceedings of the 5th IFAC International Conference on Fieldbus Systems and their Applications (FeT 2003), Aveiro, Portugal*, 2003.

[7] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on UPPAAL", in M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, September 2004, pp. 200–236. Springer–Verlag.

[8] G. Rodríguez-Navas, J. Proenza, and H. Hansson, "Using UPPAAL to Model and Verify a Clock Synchronization Protocol for the Controller Area Network", *Proc. of the 10th IEEE International Conference on Emerging Technologies and Factory Automation, Catania, Italy*, 2005.

[9] K. Etschberger, *Controller Area Network*, IXXAT Press, Weingarten, 2001.

[10] I. Broster and A. Burns, "Timely Use of the CAN Protocol in Critical Hard Real-time Systems with Faults", *Proceedings of the 13th Euromicro Conference on Real-time Systems, Delft, The Netherlands*, 2001.

[11] L. Almeida, P. Pedreiras, and J. A. Fonseca, "The FTT-CAN Protocol: Why and How", *IEEE Transactions on Industrial Electronics*, vol. 49, no. 6, 2002.

[12] J. Ferreira, L. Almeida, E. Martins, P. Pedreiras, and J. Fonseca, "Components to Enforce Fail-Silent Behavior in Dynamic Master-Slave Systems", *Proceedings of the 5^{th} IFAC Int. Symposium on Intelligent Components and Instruments for Control Applications*, 2003.

[13] I. Broster and A. Burns, "An Analysable Bus-Guardian for Event-Triggered Communication", *Proc. of the 24th Real-Time Systems Symposium, Cancun, Mexico*, 2003.

[14] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Press, 1997.

[15] H. Pfeifer, D. Schwier, and F. v. Henke, "Formal Verification for Time Triggered Clock Synchronization", *Proceedings of the 7th IFIP International Conference on Dependable Computing for Critical Applications, San Jose, CA*, 1999.

[16] J. Rufino, P. Veríssimo, and G. Arroz, "A Columbus' Egg Idea for CAN Media Redundancy", *Digest of Papers, The 29th International Symposium on Fault-Tolerant Computing Systems*, 1999.

[17] J. Charzinski, "Performance of the error detection mechanisms in CAN", *Proceedings of the First International CAN Conference (ICC94), Mainz, Germany*, 1994.

[18] K. Tindell, A. Burns, and A. J. Wellings, "Calculating controller area network (CAN) message response time", *Control Engineering Practice*, vol. 3(8), pp. 1163–1169, 1995.

[19] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues, "Fault-tolerant broadcasts in CAN", *Digest of papers, The 28th IEEE International Symposium on Fault-Tolerant Computing, Munich, Germany*, 1998.

[20] J. Proenza and J. Miro-Julia, "MajorCAN: A modification to the Controller Area Network to achieve Atomic Broadcast", *IEEE Int. Workshop on Group Communication and Computations. Taipei, Taiwan*, 2000.

[21] A. Olivero, J. Sifakis, and S. Yovine, "Using Abstractions for the Verification of Linear Hybrid Systems", in *Proc. of the 6th Int. Conference on Computer Aided Verification*, number 818 in LNCS, June 1994. Springer–Verlag.

[22] G. Rodríguez-Navas, J. Proenza, and H. Hansson, "An UPPAAL Model for Formal Verification of Clock Synchronization over CAN", Technical Report A-2-2005, Universitat de les Illes Balears, 2005.

[23] J. Ferreira, A. Oliveira, P. Fonseca, and J. Fonseca, "An Experiment to Assess Bit Error Rate in CAN", *Proceedings of the 3rd International Workshop on Real-time Networks, Catania, Italy*, 2004.