Formal Verification of a TDMA Protocol Start-Up Mechanism[1]

Henrik Lönn
Department of Computer Engineering
Chalmers University of Technology
412 96 Gothenburg, Sweden
hlonn@ce.chalmers.se

Paul Pettersson
Department of Computer Systems
Uppsala University
Box 325, 751 05 Uppsala, Sweden
paupet@docs.uu.se

Abstract

This paper presents a formal verification of the start-up algorithm of the DACAPO protocol. The protocol uses TDMA (Time Division Multiple Access) bus arbitration. It was verified that an ensemble of four communicating stations becomes synchronized and operational within a bounded time from an arbitrary initial state. The system model included a clock drift corresponding to $\pm 10^{-3}$. The protocol was modeled using a network of timed automata, and verification was performed using the symbolic model checker UPPAAL.

1. Introduction

Distributed real-time systems are increasingly used in embedded applications, many of them safety-critical systems, such as cars, aircraft and industrial robots. The computer architecture of such systems must meet several stringent requirements in terms of cost, reliability, testability, etc., and has therefore recently been the object of much research.

One crucial component of the distributed architecture is the communication system. This is the backbone of the system and has a large effect on its quality both in terms of performance and reliability.

Assuming a broadcast bus as the communication media, TDMA-based protocols have several advantages [5]. They are particularly suitable for safety-critical architectures because they facilitate clock synchronization without any message overhead [2] and support timely fault detection. In TDMA protocols, each node has a time slot where it has exclusive access to the bus. Collisions are thus avoided without the frame overhead incurred when using contention-based protocols or the token recovery algorithms needed in token-based protocols.

A characteristic of TDMA protocols is that clocks must be synchronized to guarantee collision-free broadcasts — two nodes may otherwise use the same time slot for transmission. On the other hand, messages must be broadcast to do clock synchronization.

Since we consider a multi-master system in which all nodes are equals, it is not possible to allow nodes to wait for initialization messages from a single master. We also believe that the use of a unique "jam" signal for synchronization [6] would make it difficult to detect and isolate a node that erroneously attempts to perform resynchronization.

The start-up algorithms of the TTP [4] and DACAPO [9] TDMA protocols have informal descriptions in their respective references. The latter paper also presents simulations that give an idea of the worst case duration of the start-up phase.

In several applications, the reliability of the real-time system must be better than 10^{-8} failures per hour [11]. This failure probability is so low that validating it by means of simulation is exceedingly time-consuming and yet provides only a probabilistic measure of correctness. The start-up algorithm of communication should therefore be verified using formal methods.

This paper presents a rigorous description of a TDMA start-up algorithm using timed automata. The algorithm is then verified for a system consisting of four computers, using symbolic model checking of a network of timed automata representing the computer ensemble and the bus. It is verified that all stations were synchronized within a certain deadline. The UPPAAL [7] tool was used for this purpose.

The paper starts with a brief presentation of the DACAPO protocol and an informal description of how synchronization is performed. Section 3 presents the notion of timed automata and a formal model of the protocol. Section 4 gives details on the verification of the protocol, and conclusions are presented in section 5.

[1] This work was sponsored by Volvo Research Foundation, NUTEK (Swedish Board for Technical Development), ASTEC (Advanced Software TEChnology) and TFR (Swedish Technical Research Council)

2. Protocol Description

This section briefly describes the protocol and the real-time architecture for which it is designed. In particular, the start-up algorithm is described.

2.1 General

DACAPO, Dependable Architecture for Control of Applications with Periodic Operation [10], is a conceptual computer architecture for safety-critical distributed real-time systems. The concept covers the complete computer architecture, but we will focus here on the communication protocol.

The DACAPO protocol is intended for physically small distributed systems. The bus length is limited to tens of meters to avoid problems with large propagation delays, and the number of stations is less than about 40 for reliability reasons. Although the DACAPO concept prescribes a duplex bus to meet reliability requirements, this paper considers only operation on a single bus.

The DACAPO protocol is controlled by time only. Time is divided into equally sized time slots, called *TDMA slots*, that are long enough for one message. An operational node broadcasts one message in its own TDMA slot. To minimize hardware complexity and allow simple error detection, time slot i belongs to node i, i. e. the first time slot belongs to node 0, the next to node 1, etc. up to the highest node id. When all nodes have broadcast a message, a so called *communication cycle* is completed. The communication cycle is repeated continuously during system operation, see Figure 1. The node sending order is the same in each communication cycle, but the data contents of a node's message usually change.

Each node thus needs a clock that indicates the duration of each TDMA slot and controls the transmission and reception of bits. We call this the *Bit Clock*, since it is synchronous with the bit stream broadcast on the bus. The Bit Clock will require relatively frequent adjustment owing to clock drift between nodes.

Bits are NRZ encoded (non return to zero), i. e. each bit is transmitted as either a high or low signal. NRZ encoding reduces both bandwidth requirements and noise emission as compared with encoding techniques with guaranteed transitions (such as Manchester encoding). There is no

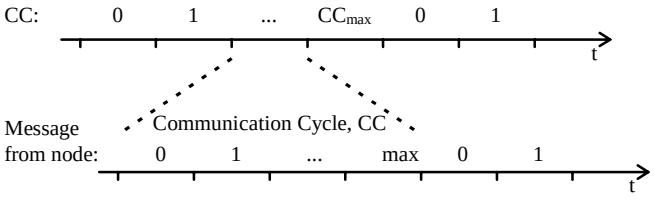


Figure 1. The Cyclic operation of DACAPO

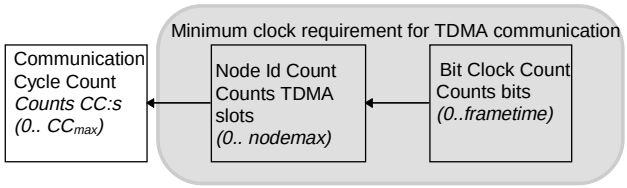


Figure 2. Time representation in each station

continuous clock adjustment during the reception of a message. Since the messages exchanged in a control application are short – a requirement to limit control delays and allow a sufficiently high sampling frequency of the control loops – adjustments during frame reception can be avoided.

A *Node ID Counter* in each node is used to keep track of the owner of each TDMA slot. The Node ID Counter is incremented each time the bit clock completes one TDMA slot interval.

A minimum requirement of the clock in a TDMA system is shown in Figure 2. The non-shaded *communication cycle count* is not used for the initialization of communication and is not considered further.

2.2 Bit Synchronization

With TDMA communication, the arrival time of a message can be used as a clock reading. Since communication is pre-scheduled, the local time of the sending node is implicitly known [2]. DACAPO uses the Daisy-Chain synchronization method [8]. With Daisy-Chain synchronization, the local clock is adjusted on each message arrival. Since nodes broadcast consecutively according to the TDMA schedule, they take turns in synchronizing the system in a Daisy-Chain manner. To avoid partitioning of the system into unsynchronized cliques, only nodes that are synchronized with at least half of the ensemble may transmit messages. Correct synchronization is decided on the basis of successful reception of a message.

To achieve fault tolerance, a *reception window* is used. Only messages whose *start of frame* field, SOF, arrives within a narrow interval are accepted (Figure 3). The size of the reception window equals the length of the SOF field.

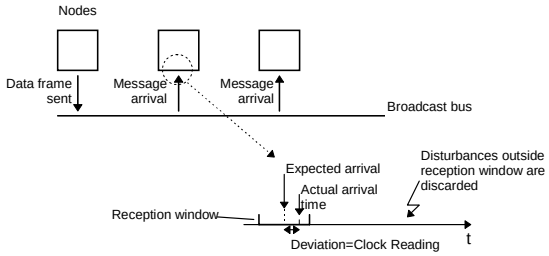


Figure 3. Exchange of clock readings.

2.3 TDMA Time Slot Synchronization

Synchronizing the TDMA time slots is equivalent to synchronizing the Node ID Count part of the local clocks. To do this, messages with the sender's Node ID Count must be broadcast, which is a problem if bus synchronization has not yet been established. A carefully designed procedure for initial synchronization of the TDMA time slots is thus necessary.

We use a state vector to describe the components of the internal state of the node that are relevant for the bus synchronization:

{Bitclock count, Node ID count, Frame Error count, Mode}

The first two items correspond to the local opinion on system time. The *Error Count* is increased each time an erroneous frame or an empty time slot is detected. It is decreased every time a message is received correctly. *Mode* is one of three protocol modes local to each node: *Normal, Resynchronization* or *Recover*.

Node operation in the different internal *Modes* is as follows:

1. **Normal Mode:**
   A. Broadcast frames according to the TDMA schedule.
   B. Assign the Node Id Count of each correctly received frame to the Node Id Count of the node.
   C. Increment Node Id Count by one in case of empty TDMA slot or erroneous frame reception
   D. Enter Resynchronization mode if messages from less than half of the nodes ($\lfloor n/2 \rfloor$ out of n) have been received correctly.
2. **Resynchronization Mode:**
   A. Be Silent.
   B. Assign the Node Id Count of each correctly received frame to the Node Id Count of the node.
   C. Increment Node Id Count by one in case of empty TDMA slot or erroneous frame reception
   D. Enter Normal Mode as soon as messages from half of the nodes ($\geq \lfloor n/2 \rfloor$ out of n) have been received correctly.
   E. When bus has been completely silent for one Communication Cycle: Enter Recovery Mode
3. **Recovery Mode:**
   A. Wait for the node's own TDMA slot and broadcast one frame if the node is part of the *recovery set*. If the broadcast would occur in the first slot in Recovery Mode, it is postponed until the next TDMA slot belonging to the node.
   B. Assign the Node Id Count of each correctly received frame to the Node Id Count of the node.
   C. Increment Node Id Count by one in case of empty TDMA slot or erroneous frame reception
   D. Enter Normal Mode when messages from half of the nodes ($\geq \lfloor n/2 \rfloor$ out of n) have been received correctly.
   E. When one frame has been sent: Enter Resynchronization Mode. If a collision was detected: reset the local Node Id Count.

In all modes, Node ID Count is set to the ID of the sending node on successful reception of a message. This is reasonable, since it was possible to transmit the message without collisions, and the sending node is thus likely to have the correct time.

The condition in 3A that a node may not broadcast in the first TDMA slot of recovery mode prevents a situation in which a fast node breaks bus silence before all nodes have seen a full Communication Cycle without bus activity. This would prevent the transition from Resynchronization Mode to Recovery Mode in these nodes. Only the first node would broadcast a message, and no node would receive enough messages to enter normal mode.

The broadcast in 3A may result in an infinite series of collisions for certain system sizes if all nodes broadcast in recover mode. To avoid this, only members of the *recovery set*, a subset of all nodes, may send a message. The nodes that will be members of the recovery set are selected before run-time.

In 3E, the Node Id Count is reset if a collision occurred to avoid a collision between the same stations if the start-up fails and a second transmission in recovery mode is necessary. Together with the recovery set limitation, this makes possible the completion of the start-up procedure without repeated collisions.

3. Formal Description of the Protocol

Our formal description of the protocol is written in the model of timed automata [1]. As the UPPAAL verification tool will be used to verify the protocol, we use the extended model of timed automata with data variables adopted in the tool.

The following gives a brief explanation of the model. We refer the reader to [7] for a thorough explanation of the model.

3.1 The Model

A *timed automaton* is a (non-deterministic) finite state automaton composed of edges and vertexes and extended with real-valued clocks and integer variables. Clocks proceed at the same rate and measure the time since they were reset. Integer variables are finite domain variables with rate zero.

In a timed automaton, the unconditional transition of a finite state automaton is extended to a triple consisting of a guard, a synchronization label and an assignment. The *guard* is a conjunction of simple constraints on the form: $v \sim n$ where v is a variable, n is a natural number (or zero), and ~ is one of $<, \leq, =, \geq, >$. An edge is said to be *enabled* if its guard is satisfied by the current clock values. An *assignment* is in the form: $x := n$ or $i := i + k$ where x is a clock, i is an integer variable and k is an integer constant. A transition's *synchronization label* is either omitted or in the form *a!* or *a?* where *a* is the name of a channel and *a!* is the complement to *a?*.

A *network* of timed automata is a collection of timed automata connected with synchronization channels. The *state* of the network is described by a continuous part representing the clock values and a discrete part representing the location of the automata and the values of the integer variables. The continuous part of the state changes as model time progresses. The discrete part may change if a transition is enabled and does not have a synchronization label. Alternatively, a hand-shaking synchronization between a pair of automata may take place if their transitions are enabled simultaneously and they have complementary synchronization labels.

Various ways of enforcing discrete transitions in the network are provided. Automata locations may be labeled with *invariants* (in e. g. Figure 6, the name of the location is marked with a simple clock constraint enclosed within parentheses) that require discrete transitions to be taken within a certain time bound. Locations may also be labeled as *committed* (marked with the prefix **c:** in the location name), requiring a discrete transition involving the automaton to be taken immediately [3]. Finally, channels may be declared as *urgent*, requiring processes to synchronize as soon as possible, i. e. when the relevant transitions are enabled [7].

3.2 Assumptions

Our model is based on the following assumptions regarding the bus communication:

- Clock precision among the station clocks is +/- 10^{-3}
  The DACAPO protocol is intended for systems with low precision oscillators, and the model must therefore contain a certain clock drift.
- A message is always received correctly if it arrives within the reception window and no collision occurs.
  We have assumed a clock drift that is sufficiently low to prevent a sender and receiver from drifting apart during message reception.
- There is no propagation delay on the bus
  In a TDMA system that is scheduled before runtime, the propagation delay for each message can be calculated before runtime. Since the sender is known, compensation can be made for the propagation delay before the local clock is adjusted.
- Broadcast messages are never corrupted unless two or more messages collide
  We do not assume any transient or permanent communication faults. Disturbances on the bus, for example, may have caused the initial loss of synchronization, but we assume that there are no further faults during the execution of our model.

3.3 The system model

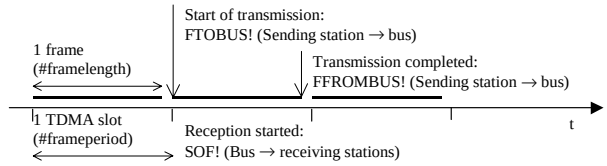


Figure 4. Successful transmission and reception.

During system operation, messages are sent and received over a broadcast bus. The main components of the system model are a bus automaton and several station automata. The stations represent computer nodes in the distributed system.

When a message is broadcast, the sender synchronizes with the bus automaton (see Figure 6) over the channel FTOBUS (see Figure 4). This puts the bus automaton in a state in which it permits synchronization with any receiver through the SOF channel to model the reception of the start of frame field.

If collisions occur, the transmitters following the first one synchronize with the bus over the JAMTOBUS channel (See Figure 5).

The JAM channel is used to model reception of corrupted messages, either as the result of a collision or because the start of frame was missed.

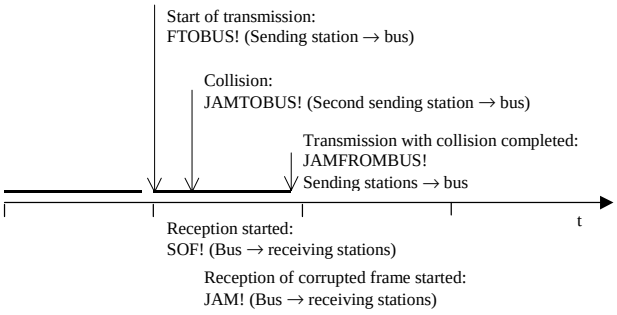


Figure 5. Message transmission with collision

The presence of a message on the communications bus is indicated by the variable *busid*. It takes the value #noid if the bus is idle or if a collision has occurred, or #0, #1, ... #nodemax depending on which node sent the message. A receiver must see a valid *busid* at the end of a message reception; otherwise, the message is considered corrupted.

The bus automaton is depicted in Figure 6. The Appendix (Table 2) gives a key to variables and constants used.

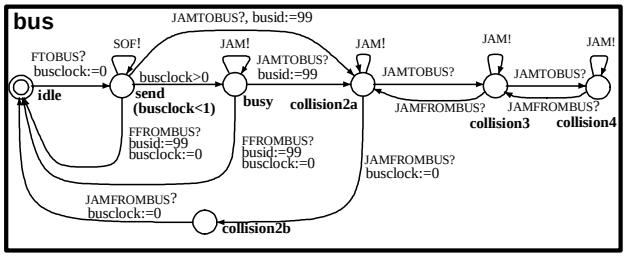


Figure 6 Timed Automaton describing the bus

3.4 The station automaton

The station automaton models the protocol behavior in each computer node. Figure 9 contains the complete automaton annotated with labels on the important transitions described in Section 2.3. Figure 9 contains the complete automaton.

The automaton is divided in three areas, belonging to Normal, reSynchronization and Recover mode, respectively. Location names in these modes are identical where applicable, but end with N, S, and R to distinguish between them. Clock variables, integer variables, and constants used in the automaton are listed in Table 2. In the description below, identifiers are indicated according to **state**, *variable*, and #constant depending on type.

To model clock drift, each guard containing a clock variable is transformed to an interval corresponding to the size of clock drift. This is done automatically by the tool. Because the model checker investigates all possible scenarios, a transition may be taken both early (fast clock) and late (slow clock) in the interval.

Before the protocol begins to execute, *bitclock* and *idcount* are initialized to random values. *Bitclock* is initialized by allowing the automaton to take its first transition at any time in the interval [0..#frameper]. *idcount* is initialized by an external initialization automaton.

A station waits for any bus activity in location **Open**. It waits for #winsize time units in Normal mode and for the duration of the entire TDMA slot in the two other modes. When a synchronization with the bus is done, the *bitclock* is reset to agree with the arrival time of the bus activity, and the counter *silence* is reset.

If a SOF synchronization is done, the station enters **Receiving** for #framesize time units. If no collision has occurred after this time, the *busid* equals the id of the sending node and *idcount* is assigned this value. If a JAM synchronization occurs or an initiated message reception fails, *idcount* is temporarily left unchanged.

If it is not possible to receive a message, *bitclock* is reset and *idcount* is incremented at the end of the TDMA slot. The error counter *errcount* is also incremented and, if no bus activity has occurred, *silence* is incremented.

When the various counters have been incremented at the end of a TDMA slot, the station may change mode. Mode changes to Normal mode are made when *errcount* falls to #errmax. In Resynchronization mode, a mode change to Recover mode is done when *silence* reaches #silencetime. Unless it was possible to reach normal mode, the station goes back to reSynchronization mode after the transmission of one frame.

At the end of the TDMA slot, a decision on message transmission is made as well. In Normal mode, a station enters **send** if *idcount* indicates that it owns the next TDMA slot. In Recover mode, **send** is entered only if *silence* is zero as well. This guarantees that a message is not sent in the first TDMA slot in Recover mode, since *silence* is not reset until after the first slot.

4. Verification

A system consisting of four stations connected to a communication bus was modeled to verify the correctness of the start-up protocol. Each station was modeled as an instance of the station automaton (i.e. gci_0, gci_1, etc*)*. Station 3 was not permitted to broadcast in recover mode (it was not part of the recovery set), since this would cause repeated collisions. The bus automaton was used to model the communication bus. In addition, a test automaton was included to support verification.

The protocol was validated and verified using the UPPAAL tool-box [7]. The model checker in UPPAAL allows for verifications of invariant and bounded-liveness properties of networks of timed automata. An *invariant* property is in the form “p is always true” and may be used to verify that certain unexpected situations never occur, e.g. “automaton A will never reach location **bad**”. A *bounded-liveness* property of the form “p is guaranteed to hold within time t” may be used to verify that an expected situation occurs within a specified time bound. This property can be verified by including the state of a test automaton in the invariant expression. For example, if automaton test transits from state **initial** to **expired** at time t, the bounded-liveness property would be “automaton test in location **expired** implies that automaton A is in location **ready”**.

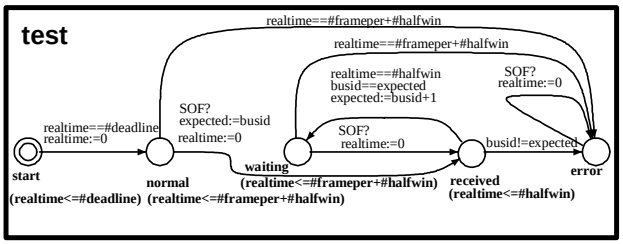


Figure 7. Test automaton used to express correctness properties.

4.1 Correctness Properties

The main correctness property of the start-up protocol requires all stations to enter normal mode within a bounded time (#deadline time units). To conveniently express this property, an auxiliary integer variable *n* was used. *n* is incremented when a station enters normal mode (i.e. on edges 2D and 3D in Figure 9) and decremented when it exits (i.e. on edge 1D). A test automaton that transits from **start** to **normal** at time #deadline was also included, see Figure 7.

Formula 1 requires all stations to be in normal mode when test has left **start**, i. e. at time #deadline:

$$\text{Inv}\ (\ (\ \text{not test.}\mathbf{start}) \rightarrow (\ n==4\ )\ ) \qquad (1)$$

We must also verify that the protocol operates correctly in Normal mode. To do this, the test automaton has a state **error** that is entered unless there is a bus transmission once every TDMA slot, and the broadcast order is 0, 1, 2, 3, 0, 1, 2, 3, etc. Formula 2 expresses the correctness property that test never reaches **error**.

$$\text{Inv}\ (\text{not test.}\mathbf{error}) \qquad (2)$$

Both these formulas were verified to hold. Moreover, the tool verified that the system never reached a deadlock, a check that is necessary to make Formula 2 meaningful. Otherwise, it could have been the deadlock, rather than correct operation of the protocol, that prevented the test automaton from reaching test.**error**.

The verification consumed a great amount of time and memory. Table 1 summarizes the resource requirements on a SUN UltraSparc II 250MHz. With 5 stations, 2 Gb of memory was not sufficient, even when no clock drift was included.

	Memory usage	Execution Time
3 stations, no clock drift	5 Mb	50s
3 stations, clock drift	10 Mb	190s
4 stations, no clock drift	83 Mb	640 min
4 stations, clock drift	541 Mb	8900 min

Table 1. Resource usage for verification

4.2 Duration of Start-Up

The lowest upper bound on #deadline was established by iterating the verification with increasing #deadline until (1) and (2) were satisfied. The worst case scenario (the largest #deadline where (1) did not hold) occurred when there was a collision between two nodes during the first transmission attempt, see Figure 8. After the collision, the duration of the TDMA slot was longer than nominal for nodes 2 and 3. This is because, in the worst case, a bit clock synchronization may occur at the end of the nominal TDMA slot when a corrupted message is present. Time was thus set back an entire TDMA slot, and the node ids of these stations were not incremented at the normal time.

The total delay until all nodes were in Normal Mode was found to correspond to about 21 TDMA slots.

5. Conclusions

We have described the start-up algorithm of a TDMA protocol for distributed systems with a broadcast bus. The start-up algorithm was verified for a system with four stations using symbolic model checking of a network of timed automata. Maximum duration of the start-up phase corresponded to 21 TDMA time slots and occurred if there was a collision on the first transmission attempt.

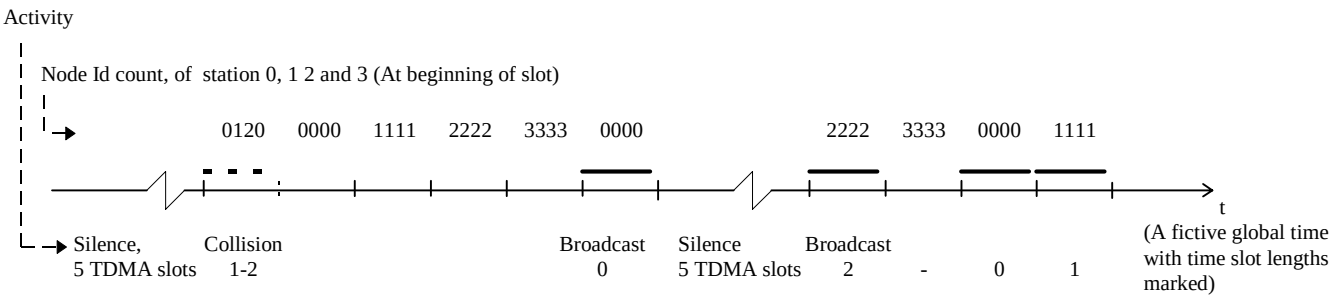


Figure 8. Worst case scenario.

The verification presented in this paper applies to four stations. Since any real system will contain more stations, this work will have to be extended. However, even the four station system is large compared with other examples verified using symbolic model checking with real-time support. Consequently, an extension to five or more nodes will be very challenging.

As regards the protocol functionality, the DACAPO protocol uses ordinary messages in the start-up phase and distributes data during this period. Although it may take several TDMA slots before all nodes have reached Normal Mode, this therefore does not mean that the communication service is unavailable for the duration of that period.

This can be compared to the Time Triggered Protocol (TTP), where a Blackout monitoring mode is entered when synchronization is lost [4]. In Blackout monitoring mode, special frames containing current time and system status are sent, but user data is not distributed. Also, if a node loses synchronization while the rest of the ensemble continues normal operation, it must wait for a *reintegration frame* to be broadcast. These are only sent periodically, as defined by the applications programmer.

Including system time in each frame as in DACAPO does cause some overhead, but single-node recovery can be done very rapidly and re-integration frames can be avoided altogether.

References

[1] Alur, R. and D. Dill: Automata for Modelling Real-Time Systems. In: Proc.of International Colloquium on Automata, Languages and Programming '90, lecture Notes in Computer Science, vol 443, Springer-Verlag, 1990

[2] Babaglou, Ö. and R. Drummond: "(Almost) no cost clock synchronization", *Proc. 17th IEEE International Symposium on Fault-Tolerant Computing*, FTCS-17, Pittsburgh, PA, USA, 1987.

[3] Bengtsson, J., W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson and W. Yi.: "Verification of an Audio Protocol with Bus Collision Using UPPAAL" In: *Proc. 8th Int. Conference on Computer-Aided Verification*. New Brunswick, New Jersey, USA, 1996. LNCS 1102, pages 244-256, R. Alur and T. A. Henzinger (Eds.).

[4] Kopetz, H. and G. Grünsteidl: "TTP–A Protocol for Fault-Tolerant Real-Time Systems", *IEEE Computer*, January 1994, pp. 14-23.

[5] Kopetz, H.: "Should Responsive Systems be Event-Triggered or Time Triggered?", *IEICE Trans. on Information and systems*, Vol. E76-D No. 11, Nov 1993, pp. 1325-32.

[6] Koopman, P. J. and B. P. Upender: " Time Division Multiple Access Without a Bus Master", United Technologies Research Center Technical Report RR-9500470, 1995.

[7] Larsen, K. G., P. Pettersson and , Y. Wang: "UPPAAL in a nutshell". *To appear: International Journal on Software Tools for Technology Transfer*, Springer Verlag, September 1997.

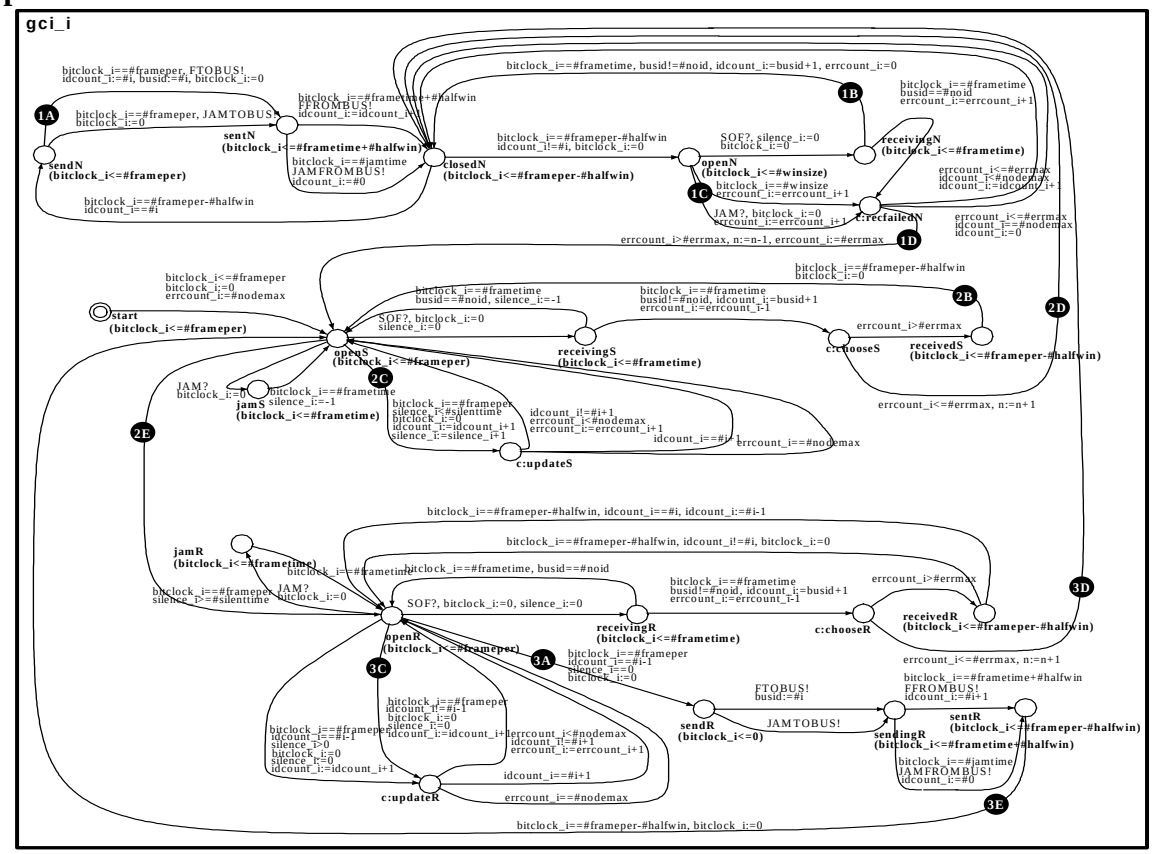
[8] Lönn, H. and R. Snedsböl: "Synchronization in Safety-Critical Distributed Control Systems", *Proc., IEEE International Conference on Architectures and Algorithms for Parallel Processing 1995*, Brisbane, Australia, 1995.

[9] Lönn, H. and R. Snedsböl: "Efficient synchronization, atomic broadcast and membership agreement in a TDMA protocol". *Proc. ISCA International Conference.on Parallel and Distributed Computing Systems*, Dijon, France, 1996.

[10] Rostamzadeh, B., H. Lönn, R. Snedsböl and J. Torin: "DACAPO: "A Distributed Computer Architecture for Safety-Critical Control Applications" *Proc. IEEE International Symposium on Intelligent Vehicles*, Detroit, MI, USA, 1995.

[11] Torin, J: Dependability in Complex Automotive Systems. Requirement Directions and Drivers. In: *Proc. Workshop on Safety and Reliability Engineering of Future Prometheus System*, Nürtingen, Germany, 1992.

6. Appendix



*Note: C*alculations involving idcount_i are made modulo #nodeant.

Figure 9. Timed Automaton describing the station.

Clocks	
bitclock_i	The local clock source for station #i
Busclock	The bus clock source. Used to enforce immediate delivery of messages
Channels	
SOF	Start Of Frame indication. Used to synchronise bit clock and start message reception
JAM	Bus activity indication. Used to synchronise bit clock.
FTOBUS	Indicates that a frame is broadcast to the bus
FFROMBUS	Indicates that a frame is removed from the bus
JAMTOBUS	Indicates that a frame is broadcast to the bus while it is busy (Resulting in a corrupted frame)
JAMFROMBUS	Indicates that a corrupted frame is removed from the bus
Integer variables	
idcount_i	node id of the owner of the current TDMA time slot
silence_i	number of TDMA time slots without bus activity
errcount_i	number of empty or erroneous messages. Increased on TDMA slots with empty or erroneous messages
busid	node id in the message on the bus
n	Number of stations in normal mode. Used for invariant expressions during verification.
Constants	
#frameper	Time between start of a TDMA slot, 224
#frametime	Length of a message frame, 218
#jamtime	Maximum duration of a tranmission of a corrupted message, 219
#nodeant	Number of nodes in the system, 4
#nodemax	The largest node id used (#nodeant-1) 3
#winsize	The size of the reception window, 2
#halfwin	The midpoint of the reception window (#winsize/2), 1
#noid	Bus id used to indicate an idle bus, 99
#errmax	Maximum number of errors tolerated in Normal mode ($\lfloor(\#nodeant-1)/2\rfloor$), 1
#silenttime	Number of silent TDMA slots before entering Recover Mode -1 (#nodeant-1), 3
#i	Local node id, in the interval [0..#nodemax].

Table 2. Clock variables, integer variables and constants used in the automata.