Automatic Recovery of the TTP/A Sensor/Actuator Network

Wilfried Steiner[1], Wilfried Elmenreich[1]

[1]Institute for Computer Engineering,
Vienna University of Technology, Vienna, Austria
{steiner,wil}@vmars.tuwien.ac.at

Abstract — *Since sensor/actuator networks are to be used in error-prone environments, it is required that media access protocols for such networks are tolerant to failures. Field studies show that the probability of transient failures to occur is much higher then the probability for permanent failures.*
After the occurrence of a transient failure, a system is in principle able to correctly execute its algorithms, however, the global system state may be unsynchronized. One approach to cope with transient failures is the implementation of automatic recovery, i. e., the system is able to reach correct operation from a faulty system state. In this paper we discuss the automatic recovery of the TTP/A field-bus protocol if the slave nodes are forced to an arbitrary position in the TDMA scheme. The analysis will be verified by using model checking techniques.

1 Introduction

Sensor/actuator networks become more and more important for real-time control applications, as for example in factory automation. TTP/A[1] is a standardized [1] approach for communication between sensors/actuators in a network that is based on a time-division multiple-access (TDMA) strategy. Prototype implementations of TTP/A networks [2, 3] were done via a hard-wired bus configuration. Due to the independence of the physical layer, wireless implementations are also possible, though not implemented currently. Sensor/acutator networks like TTP/A are to be employed in error-prone environments, where EMI interferences, cosmic rays, etc. might cause system failure.

A *failure* is usually caused by a *fault* of some component the system depends on. Faults can derive from a physical phenomenon, like the breakdown of a computer chip or from design faults, i. e., a programmer's mistake or an error in the system specification [4]. Faults can further be classified into transient and permanent faults. In our case we are regarding transient faults, which are usually related to a physical cause. As a result of a fault, a system might develop an unintended internal state, a so-called *error*. An error might cause a system to provide a service that deviates from its specification, which will be called a *failure*.

[1]Time-Triggered Protocol for SAE class A applications

In order to cope with system failures, there are two approaches:

One possibility is to mask failures by using redundant components. However, the success of this fault tolerance approach relies on the assumption coverage and the possibility to apply redundant hardware in spite of cost, weight, and power constraints.

A different approach is to implement automatic recovery of a system. Automatic recovery of a system can be seen as the appliance of the *self-stabilization* principle to fault-tolerant systems. The self-stabilization concept was introduced by Dijkstra [5], and is defined in the literature by two properties [6], *closure* and *convergence*. Self-stabilization claims that a system either stays within a closed set of legitimate, that is good, states (closure) or, if the system is in an illegitimate (bad) state, it will transit to a legitimate state within an upper bound in time (convergence). According to [7], we define the state of a system at any given time as the information needed to determine the behavior of the system from that time on.

Obviously, if the self-stabilization concept is applied to computer-controlled applications, the set of illegitimate state has to be closed as well, since it is rather uncertain that computer systems can be built that tolerate arbitrary damage.

Since the probability of multiple transient faults is much higher than the probability of permanent faults [8, 9, 10, 11], we focus on transient disturbances and discuss the abilities for automatic recovery of a TTP/A cluster after multiple transient failures.

Assume a TTP/A system in which an arbitrary number of components is affected by some external fault. After the fault disappears, the components will in fact be able to perform their algorithms correctly, but their internal states may become arbitrary and, thus, synchronization between the components may be lost. To ensure correct system operation after such transient disturbances, the disharmony has to be detected first and, in a second step, algorithms have to be executed to correct the system. A simple but expensive way to correct the system state is a system restart. However, in certain cases there is no other solution.

As a time-triggered protocol, TTP/A shows a periodical behavior where the critical elements of the system state at any given time do not influence the future states of the system after the horizon of one period. This enables the system to recover a correct state and thus transit to a legitimate state within an upper bound in time.

Although the TTP/A protocol does not have explicit fault-tolerant mechanisms, the robust behavior due to its inherent automatic recovery makes the low-cost TTP/A protocol interesting for use in dependable systems.

The objective of this paper is twofold. First we argue that a TTP/A system will converge towards a correct operation mode if the slave nodes start at an arbitrary position in the TDMA scheme. Second we explore the applicability of the model-checking tool UPPAAL to verify our assumptions.

Related work was done in [12] where the automatic recovery mechanisms of the TTP/C protocol where discussed. [13] discusses the self-stabilizing property of the TTP/C group membership algorithm and gives model-checking experiments.

This paper is structured as follows: in Section 2 we discuss, how self-stabilization can be seen as a form of fault-tolerance in general. Then, in Section 3 we give an overview of the TTP/A protocol. Section 4 gives the definition of a correct protocol execution as well as the self-stabilization assumption. The formal verification process is described and

results are listed in Section 5. Finally, we conclude in Section 6.

2 Self-Stabilization as Automatic Recovery

Figure 1, taken from [12] depicts a generic concept of automatic recovery. In the center, the safe system states are depicted, including states in which failures are masked. The rings depict unsafe system states, and we say the more outward the ring the less requirements for a safe system state hold true. Following this idea, three scenarios are shown in Figure 1, *a*,*b*,*c*. Scenario *a* depicts a minor fault while scenario *b* shows a more serious fault. Scenario *c* denotes the occurrence of a fault where no transition to a safe system state is guaranteed, although, the system may reach a safe state by chance. Furthermore, the figure shows the execution of recovery algorithms towards a safe system state (dotted line).

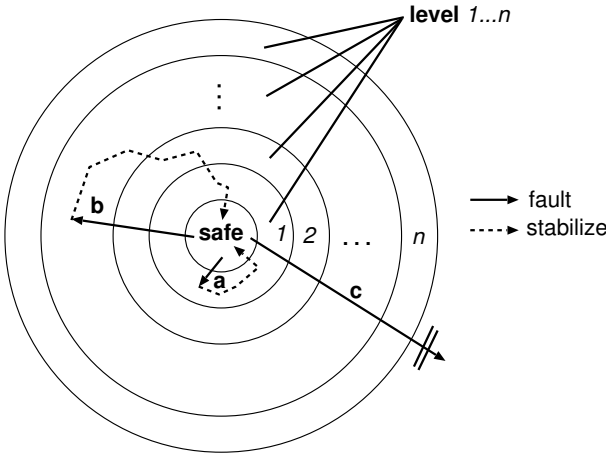

Figure 1: Stabilization in a safety-critical system

Automatic recovery usually requires a *failure detection* phase as well as a *failure correction* phase. The failure detection phase detects if the system is in a bad state and the consecutive execution of failure correction algorithms ensure a transition back to a safe system state.

Definition 1. *We call a system self-stable if it provides mechanisms for automatic recovery.*

3 Overview on TTP/A

TTP/A is a time-triggered protocol used for the communication of one active master with or among smart transducer nodes within a cluster. This cluster is controlled by the master, which establishes a common time base among the nodes. In case of a master failure, a shadow master can take over control. Every node in this cluster has a unique alias, an 8 bit (1 byte) integer, which can be assigned to the node a priori or set at any time via the configuration interface.

The TTP/A communication is organized into rounds. Figure 2 shows a round sequence of four subsequent multi-partner (MP) rounds seperated by inter round gaps (IRG). IRGs

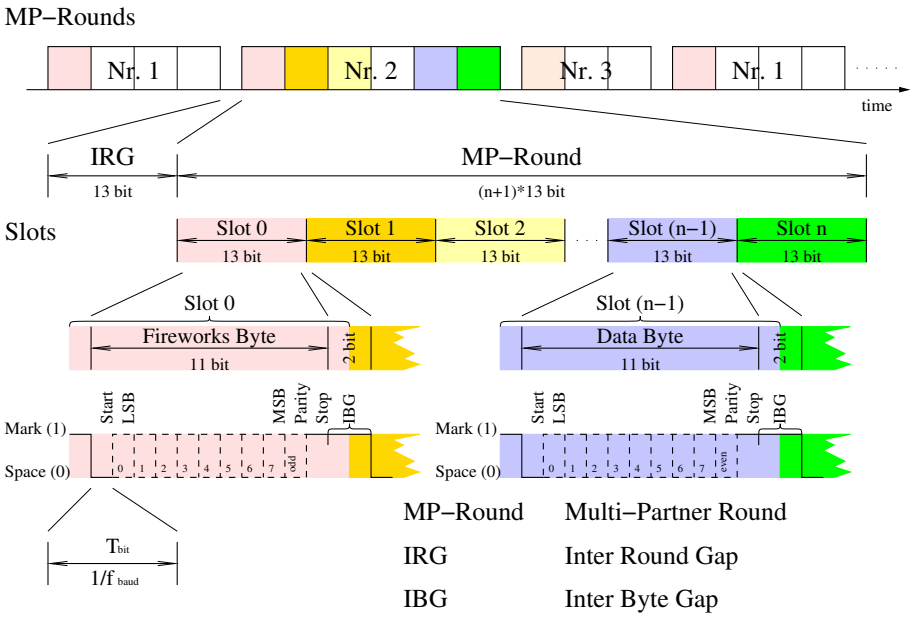

Figure 2: TTP/A Communication Layer

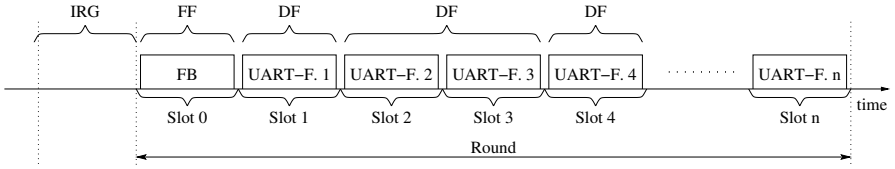

Figure 3: Layout of a TTP/A Round

are slots where the TTP/A bus is inactive for at least 13 bit cells. A TTP/A round consists of one or more frames. A frame is a sequence of bytes transmitted by one node. A byte is transmitted in a slot consisting of 13 bit cells (one start-bit, eight data-bits, one parity, one stop-bit and a two bit cell wide inter byte gap (IBG)).

The rounds are independent from each other. Every round starts with a fireworks frame (FF) sent by the master. The arrival of the fireworks frame is a synchronization event for every node in the cluster and identifies the round. According to the specification of the selected round, the fireworks frame is followed by data frames (DF) of specified length from the specified nodes. Each such frame is described by an entry in the round descriptor list (RODL) in the file-system of the sender and the receiver(s).

Because the slot position at which each communication action takes place is defined a priori, no further communication for bus arbitration is necessary. Figure 3 shows the layout of a TTP/A round.

3.1 Data Transmission

For the transmission of bytes on the TTP/A bus, a standard UART format has been chosen (see figure 4): One start bit, 8 data bits, one parity bit and one stop bit. The parity for data

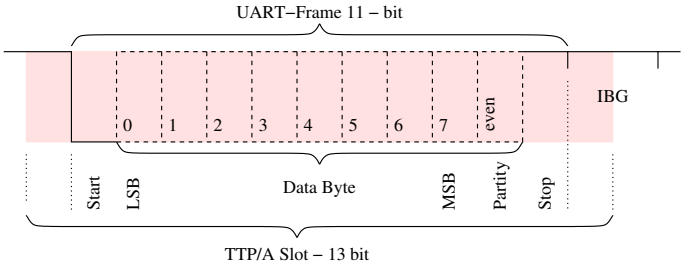

Figure 4: Data Byte

bytes has to be even, whereas for the fireworks byte the parity must be odd. The passive state on the bus is logical 1 (high). The start of a new byte is marked by the falling edge of the start bit. The stop bit (logical 1) is followed by the inter byte gap which is, in the current implementation of TTP/A, 2 bit cells long, for which the bus is also in passive state (logical 1). So the 11 bit cells long UART frame is embedded in a TTP/A timeslot of 13 bit cells times.

The length of the inter byte gap (IBG) depends on the chosen baud rate of the network and its physical size. In faster and more expansive TTP/A networks the length of the IBG might be up to 5 bit cells long.

A new round always is started by a *fireworks byte* (FB). The FB is transmitted with odd parity, in contrast to other data bytes which are sent with even parity. There are only 8 valid fireworks bytes (6 for multi partner rounds and 2 for master/slave rounds; see table 1). The fireworks bytes (protected by a parity bit) have Hamming distance of at least 4.

Firework	Meaning	Description
0x78	RODL=0	Multi-Partner Round 0
0x49	MSD	Master/Slave Data Round
0xBA	RODL=2	Multi-Partner Round 2
0x8B	RODL=3	Multi-Partner Round 3
0x64	RODL=4	Multi-Partner Round 4
0x55	MSA	Master/Slave Address Round (startup sync.)
0xA6	RODL=6	Multi-Partner Round 6
0x97	RODL=7	Multi-Partner Round 7

Table 1: Firework codes

The Master/Slave Address (MSA) fireworks byte has been designed to generate a regular bit pattern, which can be used by slave nodes with an imprecise on-chip oscillator for startup synchronization (see figure 5).

The three least significant bits (bit 0...2) of the FB denote the round name. The remaining bits (5 data and one parity bit) are used for error detection.

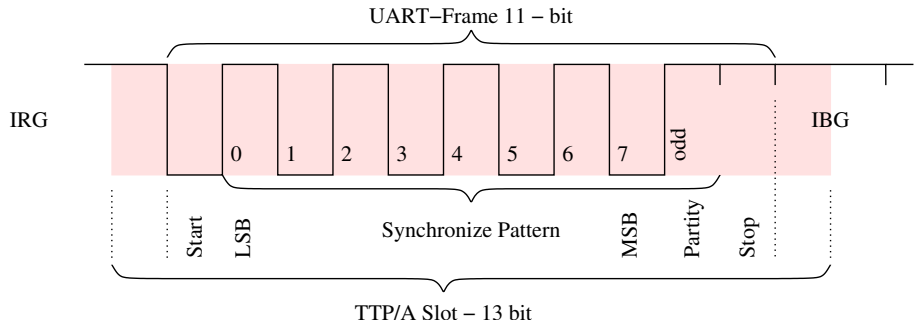

Figure 5: Synchronization Pattern

3.2 Properties of the Fireworks Byte

The generation of the firework codes had several requirements (see [14]): First, the byte `0x55` must be a part of the code because this regular bit pattern is also used for initial synchronization of the slaves' UARTs. Hamming distance should be maximized and the resistance against burst errors should be optimal. The firework bytes are all sent with odd parity and the lower three bits of the code have to be equal to the round number.

Code Generation: Because `0x55` must be a member of the code, it was impossible to use a cyclic redundant code (CRC) with the requested properties. So the code was created by using an exhaustive search method.

Hamming Distance: The occurring Hamming distances are 4, 6, and 8. So the code will detect all errors of weight less than 4.

Parity: The code includes an odd parity bit. So it will also detect all errors with an odd weight above 4.

Burst Errors: Every possible burst error will be detected by the code.

Bulk Errors (force the bus to low or high): It is impossible to get a valid FB by setting (or clearing) one or more adjacent bits. Therefore, it is impossible to corrupt a FB by applying a direct voltage impulse to the bus and get another valid FB.

Error Exposure: Due to the fact that the byte is protected by a parity bit during transmission, we have an error exposure $\geq \frac{2^9-8}{2^9} \geq 98,43\%$.

Error Pattern: There exist only 7 error patterns that will not be exposed under certain circumstances. Four of them have weight 4 (`0x02D`, `0x11C`, `0x131`, and `0x1C2`), two error patterns have weight 6 (`0x0DE` and 0x0F3), and one has weight 8 (`0x1EF`).

3.3 Types of Rounds

Multi-partner Rounds: A multi-partner round is used to transmit messages across the bus from several nodes in predefined slots. Multi-partner rounds are scheduled periodically by the master. They are used to update real-time images, supporting the real-time view of the cluster and also to periodically re-synchronize the slaves' clocks. It is possible to define 6 different multi-partner rounds per cluster.

Master-slave Rounds: The master of a TTP/A cluster can schedule master-slave (MS) rounds to read data from an IFS file record, to write data to an IFS file record, or to execute a selected IFS file record within the cluster.

The master-slave address (MSA) round specifies the node, the operation and the local address of the desired data within the addressed node. The master-slave data (MSD) round is used to transmit the data between master and slave.

Broadcast Rounds: A broadcast round is a special form of MS round were the name of the addressed node in the MSA round is set to `0x00`. All nodes except those with node alias `0xFF` are addressed by such a broadcast. Because more than one node is addressed in such a round, only write and execute operations are permitted for broadcast rounds. An example for such a broadcast is the *sleep* command, which puts all nodes in a TTP/A cluster into *sleep mode*.

4 Stabilization Assumption

In this paper we focus on the self-stabilization property of the TTP/A protocol if the slave nodes start at an arbitrary position in the TDMA scheme. Since nodes may send therefor concurrently on the communication medium, collisions can occur. However, the cyclic nature of the TTP/A protocol guarantees that the nodes will become synchronized by the master within an upper bound in time and collisions, therefor cannot occur anymore.

Definition 2. *Correct System State: A TTP/A system is in a safe system state if the nodes execute the TDMA scheme synchronously and therefor no collisions occur on the communication medium.*

Assumption 1. *A TTP/A system will converge towards a correct system state within an upper bound in time if the nodes start at an arbitrary position in the TDMA schedule.*

The proof of Assumption 1 will be demonstrated in Section 5.

5 Formal Verification

To get confidence in our assumptions we performed several model-checking experiments using the tool UPPAAL2K [15]. Model-checking is a method for formal verification which allows, once an appropriate model is created, a fully automatic verification process, without user intervention.

5.1 Model Overview

The model consists of timed automaton for the TTP/A slave, the TTP/A master, and the communication channel. If a slave reaches its sending slot, it transmits a *send* signal to the channel module. The channel module waits for the propagation delay to expire and signals a *traffic* event to all slave modules. When the slave stops sending it signals a *stop* event to the channel, which triggers a *silence* event to the slaves. Also, if the master reaches the point in time to send the fireworks byte it sends a *trigger* event to the channel which forwards the trigger to the slaves if no other slave is currently sending. Time is represented by *ticks*. To keep track of the progress of real time we use a real-time counter, called *real-time*.

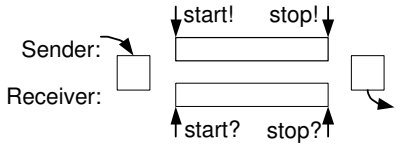

Figure 6: Frame transition model

5.2 Model Description

5.2.1 Frame Transmission Model

Each transmission of a frame is modelled in the time domain and value domain. The time domain is represented by two events, *start* and *end.* The value domain is modelled by one or more global variables. The idea is sketched in Figure 6.

5.2.2 TTP/A Slave

The timed graph of a TTP/A slave is depicted in Figure 7. The INIT state is the starting state of the model. The node has to pass immediately from this state to the ongoing state. It is free to take one of the 4 paths, which can be interpreted as the initial randomization of the slot position a node starts in. The slot position is stored in the variable *local_slot. After* I*NIT, the decision is done (*D*ECISION_POINT1) whether the current slot position matches the nodeID (*nodeID). If so, the node's sending slot is reached and the node prepares for sending (PRE_SEND). After PRE_SEND state expires, a *send! signal is triggered to the channel module and the actual* T*RANSMIT state is entered. After transmission, that is, the transmission timeout expires, the* stop! signal is triggered to the channel model, indicating that the node has stopped transmission and the POST_RECEIVE state is entered. After POST_RECEIVE expires, the decision is taken (DECISION_POINT2), whether it was the last slot in the TDMA schedule, if so, the node transits to NEW_ROUND state and waits for the fireworks byte sent by the master module that indicates the beginning of a new round. If at DECISION_POINT1 it is ascertained that the current slot position does not match the nodeID, the node receives data from the channel during the TDMA slot (RECEIVE). After the slot expires, the same state is reached (DECISION_POINT2) and the node will either transit to NEW_ROUND or to DECISION_POINT1. The *local_slot variable is incremented either during* T*RANSMIT or* R*ECEIVE state, which establishes the cyclic execution of the TDMA scheme.*

5.2.3 TTP/A Master

The timed automaton of the TTP/A master is depicted in Figure 8. The master starts in state I*NIT and transits to* N*EW_ROUND during an interval of* 1 *TDMA round. This behavior simulates a transient failure, i. e., the master starts unsynchronized to the slaves. Passing* N*EW_ROUND triggers the signal* trigger!, which simulates the sending of the fireworks byte. After the trigger, the node proceeds to the WAIT state where it remains for one TDMA round. After one TDMA round the master starts over and triggers the beginning of a new TDMA round.

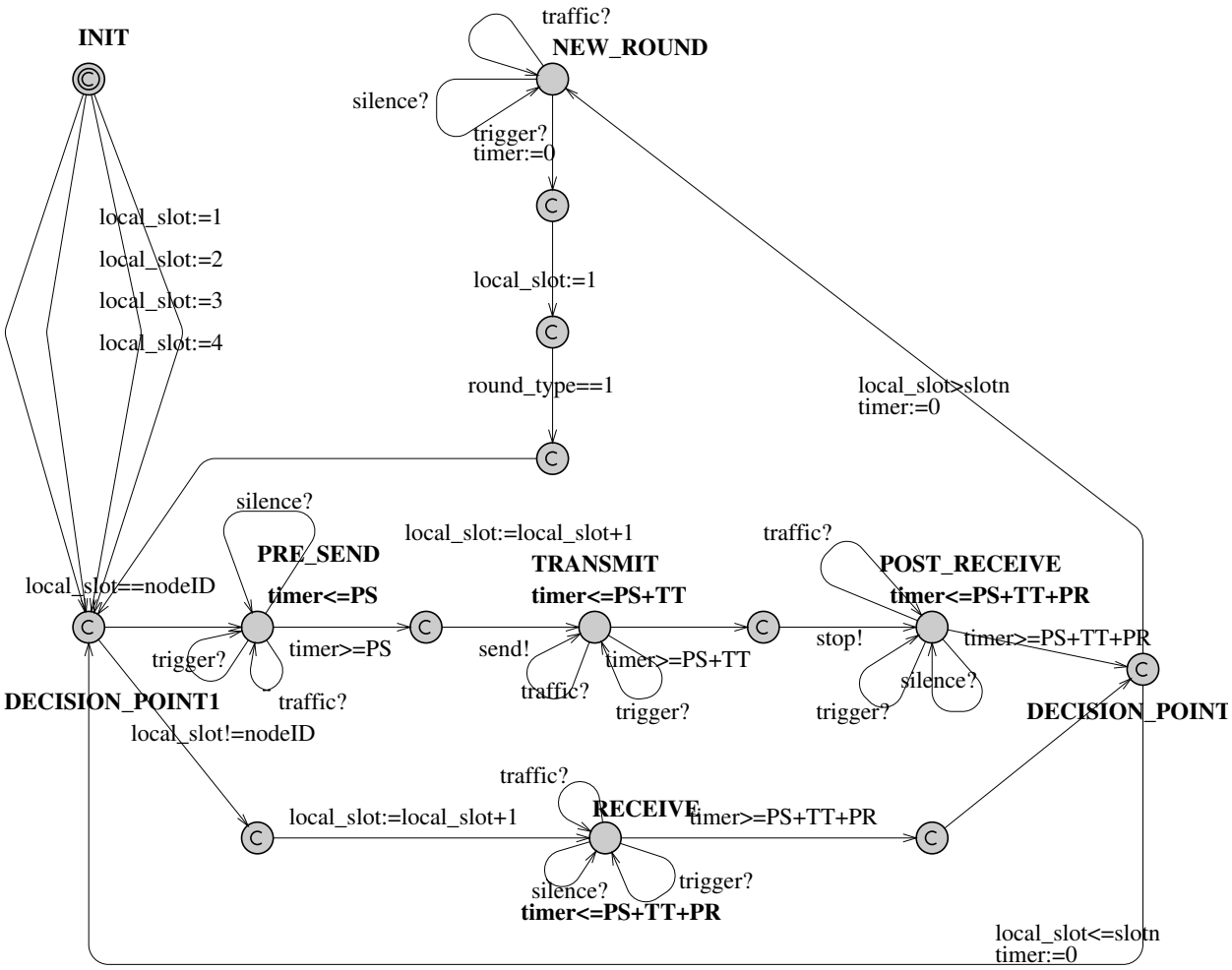

Figure 7: Model of TTP/A slave

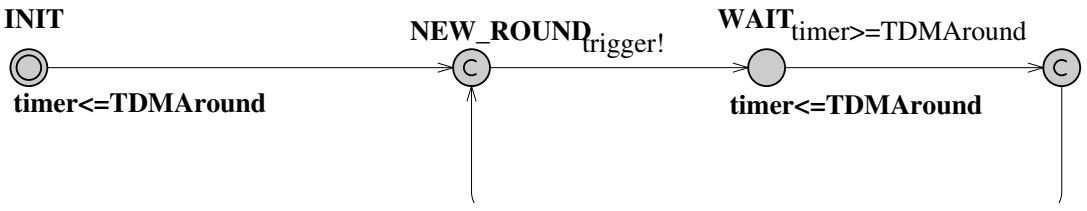

Figure 8: Model of TTP/A master

Note that the model could be sophisticated to model the fireworks byte as transmission with a *send! signal as well as a* signal! similar to the slave model. However, in its current form the model is appropriate enough to simulate the collision of the fireworks byte with regular data sent by some node.

5.2.4 Communication Channel

Figure 9 depicts the communication channel of the TTP/A network. In contrast to the other models (TTP/A master and TTP/A slave) state transitions are generally done with respect to real-time, except for the simulation of the propagation delay of the medium in state SENDING_TWO that allows collisions to happen (if this timer was neglected, collisions could simply be avoided by listening of the nodes to the medium before sending, but such a model would not be realistic). The channel model starts in state IDLE upon reception of a trigger the channel forwards this trigger to all other nodes indicating a new TDMA round. If the signal *send_node*x *is received, where x* denotes the node ID of a respective node, the node transits to state SENDING_ONE. If during the propagation delay no more *send_node*x *signal is received, no collision occurred, and the* traffic*x* signal is sent to the respective node (again x *denotes the node ID of a respective node). If, however during* S*ENDING_ONE a second* send_node*x* signal is received, the node sets the *collision variable to* 1 *and transits to state* S*ENDING_TWO. Again in this node, if no more signal is received, the channel forwards the* traffic*x* signals to the nodes, or if more signals are received, SENDING_THREE is entered. The same procedure is done for SENDING_FOUR and can be extended analogously for a higher number of nodes. If during one of the sending states the master sends a trigger signal for a new TDMA round, this signal is simply ignored by the channel module and the *collision* variable is set to 1.

Note that with this approach it is assumed that a collision of the fireworks byte with some data from a slave results in an invalidation of the fireworks byte. In the states on the right hand side of Figure 9 the channel simply waits for the *stop_node*x *signal. Again (***x** *denotes the node ID of a respective node). Upon reception of the last* stop_node*x* signal the channel sends the *silence*x *signal to the respective node indicating silence on the bus. The* collision variable is set to 0 upon entering IDLE state again, leaving the channel stateless.

5.3 Verification Process and Results

Given a UPPAAL2k model, the verification process of given properties is done fully automatically. The properties have to be specified in a logic notation. Assumption 1 is written as:

$$A\Box((real\_time > \Delta) \rightarrow (collision == 0)) \quad (1)$$

Where $A\Box$ means that the following condition holds true in all states. That is, if the real-time counter *real_time*, that simply counts the progress of time, reaches a certain value, no more collisions will occur.

Table 2 lists results of the model-checking procedure for different parameters of the model.

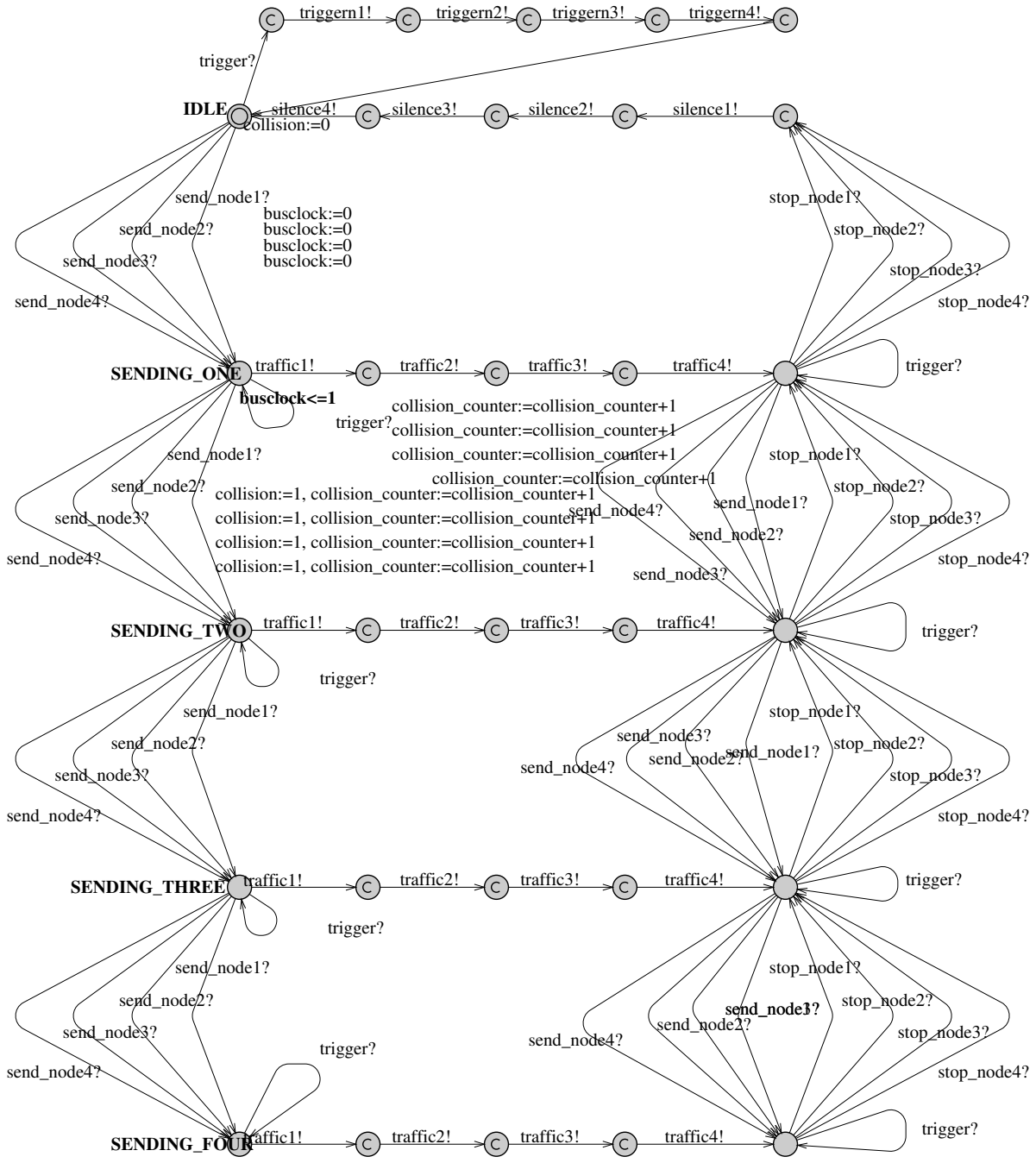

Figure 9: Model of TTP/A channel

PS	TT	PR	Δ	TDMA rounds
2	2	2	22	1 - PR
5	10	5	75	1 - PR
15	15	15	165	1 - PR

Table 2: Output of different configurations

Concluding from the model-checking results we see that our hypothesis holds. The upper bound for the stabilization with respect to collisions on the communication channel is 1 TDMA round.

Table 3 lists different assumptions that were tested against the model under various configurations to get confidence in the model's accuracy.

Assumption	Description	Evaluation
$A\Box(\neg deadlock)$	no deadlocks occur	true
$A\Box collision == 0$	no collisions occur	false
$A \diamond node1.NEW\_ROUND$	node1 reaches NEW_ROUND state	true

Table 3: Evaluation results

6 Conclusion

In this paper we discussed automatic recovery of a TTP/A network after a transient failure affecting an arbitrary number of TTP/A slaves. It was explained how TTP/A system reaches a correct system state from an arbitrary system state. This assumption was validated using a timed automata model of the protocol which was verified using the model-checking tool UPAAL.

In the future we plan to extend the model of TTP/A to master-slave rounds and perform detailed model-checking as well as exhaustive fault-injection experiments.

References

[1] Object Management Group (OMG). *Smart Transducers Interface V1.0*, January 2003. Specification available at http://doc.omg.org/formal/2003-01-01 as document ptc/2002-10-02.

[2] P. Peti and L. Schneider. Implementation of the TTP/A slave protocol on the Atmel ATmega103 MCU. Research Report 28/2000, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, August 2000.

[3] C. Trödhandl. Architectural requirements for TTP/A nodes. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[4] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.

[5] E. W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, 1974.

[6] M. Schneider. Self-stabilization. *ACM Computing Surveys (CSUR)*, 25(1):45–67, 1993.

[7] L. A. Zadeh. *The concept of system, aggregate, and state in system theory*, volume 8 of *Inter-University Electronics Series*. McGraw-Hill, 1969.

[8] E. Normand. Single Event Upset at Ground Level. *IEEE Transactions on Nuclear Science*, 43:2742–2750, 1996.

[9] T. J. O'Gorman. The Effect of Cosmic Rays on the Soft Error Rate of a DRAM at Ground Level. *IEEE Transactions on Electron Devices*, 41:553–557, 1994.

[10] J. Wilde, W. Wondrak, and W. Senske. Reliability Requirements for Microtechnologies used in Automotive applications. In *Proceedings of the Congress for Microsystems and Precision Engineering, MicroEngineering 99*, Stuttgart, Germany, October 1999. Stuttgarter Messe- und Kongressgesellschaft GmbH.

[11] B. Pauli and A. Meyna. Reliability of Electronic Control Units in Motor Vehicles. *SAE Technical Paper Series*, February 1998.

[12] W. Steiner, M. Paulitsch, and H. Kopetz. Self-stabilization in a time-triggered architecture. Research Report 21/2002, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[13] J. Rushby. An Overview of Formal Verification for the Time-Triggered Architecture. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 83–105, Oldenburg, Germany, September 2002. Springer-Verlag.

[14] W. Haidinger and R. Huber. Generation and analysis of the codes for TTP/A fireworks bytes. Research Report 5/2000, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2000.

[15] P. Pettersson and K. G. Larsen. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, 2000.