Modeling Bus Communication Protocols Using Timed Colored Petri Nets — The Controller Area Network Example

Marko Bago[1], Nedjeljko Perić[1], and Siniša Marijan[2]

[1] University of Zagreb,
Faculty of Electrical Engineering and Computing,
Unska 3, 10000 Zagreb, Croatia
{marko.bago,nedjeljko.peric}@fer.hr
http://www.fer.hr

[2] Končar - Electrical Engineering Institute,
Fallerovo šetalište 22, 10000 Zagreb, Croatia
sinisa.marijan@koncar-institut.hr
http://www.koncar-institut.hr

Abstract. Engineers in industry usually design new systems based on earlier experience and available development tools. Unfortunately, software tools that enable industrial users insight into systems' inner behavior before the production, are still not in everyday use. However, such tools, when used properly, can save both time and money.
In this paper a system based on Controller Area Network is modeled using timed colored Petri nets. This system is verified for the desired properties and then it is validated. Validation is done using two real-life vehicle control units used in light rail applications. The results, as well as possible future use of the model, are presented.

Key words: Controller Area Network, timed colored Petri net, modeling, simulation, verification, validation

1 Introduction

Distributed systems are based on communication networks. Different communicating entities (nodes) interact with each other. The interaction becomes more complex as the system grows. Developing a distributed system also means developing an adequate communication network, one that is able to support all the requirements set by each individual node.

During recent light rail vehicle development CAN was used as a fieldbus, [1, 2]. Testing the functionality of such a system required *Software Simulation Tools* (*SSTs*) and, in the end, real-life equipment. SSTs allow for easier and faster development of a system than real-life equipment. Unfortunately, communication SSTs are usually based on a single communication protocol. This raises at least two problems. First, if the system uses more than one communication protocol, it is necessary to use more than one SST. The interaction between the tools is either difficult or impossible. Second, using more than one SST increases the cost, and the development engineers have to be familiar with several different development environments.

One solution to these problems would be a single SST with support for multiple communication protocols, [3, 4]. In order to achieve this, a development environment for the SST has to be defined. This environment should fulfill the following requirements: it should be flexible, user friendly, and it should support verification and validation of communication system models.

"Whereas verification checks a model against a given specification, validation checks a model against the modeled system.", [6].

Verification of a system can only be performed if the system is modeled using some formal method.

In [5] it is correctly observed that *"A model is always a reduced rendering of the system that it represents."*. Five key characteristics for a *Model-Driven Development* (*MDD*) of software are given: abstraction, understandability, accuracy, predictiveness and inexpensiveness. These five characteristics are universal to all modeling and can be recognized in the requirements for the SST, e.g. understandability can be understood as a part of user friendly requirement, accuracy and predictiveness are in fact support for verification and validation, etc.

Petri nets fulfill all of these requirements. The graphical representation of the net gives the user an easier understanding of the modeled process, and system models can be verified because Petri nets are a formal method. For a correct validation, a real-life system should be modeled. Results obtained from the simulation model and from the real-life system should be compared. For communication protocols two factors are important for the validation: (i) order of messages gaining bus access must be identical and (ii) message propagation time has to be equal to that of the real-life system.

This paper presents a way to model *Controller Area Network* (*CAN*) communication protocol using timed *colored Petri nets* (*CPN*), [7–9]. A short introduction to the CAN bus is presented in Sect. 2. In Sect. 3 a short introduction to hardware and software tools used to create, verify and validate the model, is given. Section 4 presents the CPN model of the CAN bus created using *CPN Tools*. All modules used to create the CAN bus network are explained. Section 5 presents verification and validation methods used on the model and explains how and why they are used. This section also presents the results of the verification and validation processes. Verification was achieved using state space queries within CPN Tools. Validation of the model was done using two real-life *vehicle control units* (*VCUs*) used in light rail applications. Section 6 concludes the paper and gives a brief overview of the future work.

2 CAN bus communication

CAN was originally developed for automotive applications in the early 1980's by Robert Bosch GmbH. The CAN protocol was internationally standardized by

ISO (International Organization for Standardization) and SAE (Society of Automotive Engineers). Today, CAN is used in many markets, like motor vehicles, industrial automation, medical equipment etc.

CAN bus is an event-triggered, multi-master, bus communication system with priority-based access control and automatic retransmission of corrupted frames, [7, 8]. Frames are labeled with identifiers (11-bit or 29-bit) that are used to determine both the priority and the content of a frame. Frame identifiers (ID) are transmitted together with useful user data. There are no node addresses. Two bit levels exist on the bus, dominant and recessive. The dominant bit level (logical 0) overwrites the recessive bit level (logical 1). The CAN bus bit rate can be up to 1 Mbit/s.

CAN protocol uses four different frame formats for the communication:

1. *Data frame* - carries data from the transmitter to all receivers on the bus. Data is labeled with a message ID.
2. *Remote frame* - used to request transmission of a data frame with identical message ID.
3. *Error frame* - transmitted by any node (transmitter or receiver) in case of a bus error detection.
4. *Overload frame* - used for providing an extra delay between the preceding and succeeding data or remote frames.

Data frames and remote frames are separated from the preceding frames by an interframe space. There are two parts of interframe space, intermission and bus idle. Intermission is inserted after data or remote frame and is 3 bits (recessive) long. During intermission no node is allowed to send either data or remote frames. Only sending of an overload frame is allowed. Bus idle may be of arbitrary length and only recessive values are on the bus.

CAN frames. A CAN frame is simultaneously accepted either by all nodes or by none.
Data frames are composed of seven different bit fields, as in Fig. 1.a and Fig. 1.c: *SOF (Start Of Frame)* - 1 bit, *arbitration field* - 12 bits (11-bit ID) or 32 bits (29-bit ID), *control field* - 6 bits, *data field* - [0-8] bytes, i.e. [0-64] bits, *CRC field (Cyclic Redundancy Check)* - 16 bits, *ACK field (Acknowledge)* - 2 bits and *EOF (End Of Frame)* - 7 bits.
Remote frames are composed of six different bit fields, as in Fig. 1.b and Fig. 1.d: *SOF* - 1 bit, *arbitration field* - 12 bits (11-bit ID) or 32 bits (29-bit ID), *control field* - 6 bits, *CRC field* - 16 bits, *ACK field* - 2 bits and *EOF* - 7 bits.
Error frame is composed of two fields, as in Fig. 1.e and Fig. 1.f: *error flag field* - [6-12] bits and *error delimiter* - 8 bits.
Overload frame is composed of two fields, as in Fig. 1.e and Fig. 1.f: *overload flag field* - [6-12] bits and *overload delimiter* - 8 bits.

Frame coding. The following bit sequences (fields) will be coded by the bit stuffing method: SOF, arbitration field, control field, data field and CRC sequence. Whenever a transmitter detects five consecutive bits (including stuff

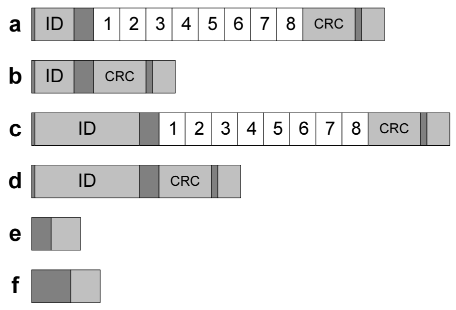

Fig. 1. Following frames are presented: **a)** 11-bit ID, 8 byte Data frame; **b)** 11-bit ID remote frame; **c)** 29-bit ID, 8 byte Data frame; **d)** 29-bit ID Remote frame; **e)** 6-bit flag field error/overload frame; **f)** 12-bit flag field error/overload frame. The data fields are colored white.

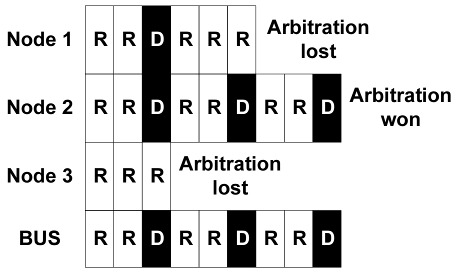

Fig. 2. An example of an arbitration process on the CAN bus. The arbitration is among three competing nodes. *Node 3* is the first to lose the arbitration, while the *Node 2* won the arbitration. *BUS* shows values present on the CAN bus.

bits) of identical value in the bit stream, it automatically inserts a complementary bit into the stream. The receiver automatically destuffs this bit from the received bit stream.

Bus access. Every node has the right to access the idle bus at any moment in time. SOF (data or remote frame) marks the beginning of bus access and contention-based arbitration takes place. Every transmitter compares the bit being transmitted with the bit on the bus. If a recessive bit is being sent, but a dominant bit is detected, the node loses arbitration and will not send any more bits, Fig. 2. The node that lost the arbitration becomes a receiver. If a data and a remote frame are sent to the CAN bus at the same time, and if the two frames have identical identifiers, then the data frame will win the arbitration process, i.e. it will gain access to the CAN bus first.

Error detection and handling. The following mechanisms are provided for error detection: *monitoring*, *stuff rule check*, *frame check*, *15-bit CRC*, *ACK check*.
Monitoring: A node sending a bit on the bus monitors the bus at the same time. If the value sent is different from the value detected then an error is detected. Exceptions happen during arbitration, during an ACK slot and while sending an error passive flag.
Stuff rule check: A stuff error is detected during the sixth consecutive bit of equal level in the frame field coded by the bit stuffing method (SOF, arbitration field, control field, data field and CRC sequence).
Frame check: This error is detected if one or more illegal bit values is detected in a bit field, e.g. during EOF the node detects a dominant bit and only 6 recessive

bits instead of 7.
15-bit CRC: CRC is calculated by the receiver in the same way it is calculated by the transmitter. If the values do not match then an error is detected.
ACK check: An ACK error is detected by the transmitter whenever a dominant bit is not detected during the ACK slot.
Errors are registered and error frames are automatically retransmitted.

In this paper 11-bit message IDs and 500 kbit/s were used for CAN communication.

3 Tools

Creation, verification and validation of a simulation model requires different tools. The tools can be divided into two groups, hardware and software tools, i.e. hardware equipment and a computer with appropriate programs. In order to create, verify and validate a timed colored Petri net model, CPN Tools is used, [10]. Validation also requires a real-life system. The real-life system was composed of two vehicle control units (VCUs) that are used in light rail applications, [2].

3.1 Hardware tools

Hardware used for validation purposes consists of three CAN communication nodes. Two CAN communication nodes are VCUs used for control of TMK2200 trams operating in the city of Zagreb, Croatia, Fig. 3, [1]. The third node is a general purpose CAN communication node connected via a USB cable to a laptop PC. It is used to start the message exchange between VCUs and to log all data traffic on the CAN bus. All logged messages are time stamped with 1 μs resolution. The hardware validation test system is given in Fig. 3.

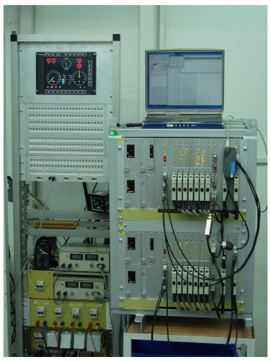

Fig. 3. The assembled test system consisting of a VCU and a PC on top of it.

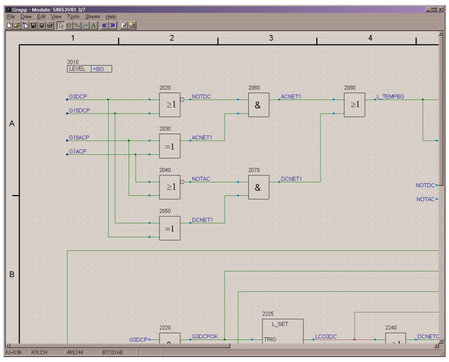

Fig. 4. The integrated development environment for the programming of the VCU.

Vehicle Control Unit. VCU is a twin-channel, multiprocessor system that supports sequencing, protection, regulation, diagnostic and communication functions, [2].

Integrated Development Environment (IDE). User programs are developed by the engineers that are application oriented. To support them, an IDE based on block diagrams was developed, Fig. 4. This IDE was used to program the validation system.

3.2 Software tools

CPN Tools software was used to create, verify and validate timed colored Petri net models. CPN Tools is a program developed and maintained by the CPN group from University of Aarhus, Denmark. The program is capable of creating hierarchical timed colored Petri net models, [10]. Petri nets are represented by the graphical layout, while additional information and interaction of the model comes from the CPN ML programming language. CPN ML is based on the Standard ML programming language, [11, 12]. Using CPN ML, it is possible to define complex data structures and functions to handle these structures.

CPN Tools is based on the formally defined syntax of colored Petri nets. The semantics, i.e. the behavior of the net, is also defined. The defined semantics enable simulation of the model. Simulation can be interactive, i.e. with user intervention, or automatic. Automatic simulation enables validation of the system. Since both the syntax and the semantics of the CPN models created using CPN Tools are defined, it is possible to generate the full state space of the models. The state space can be queried. Queries have to be written using CPN ML programming language. Queries enable verification of the desired properties of the system.

4 Model

The concept of the bus communication system is given in Fig. 5. The nodes have an identical structure. This means that it is possible to create one node structure, and reuse it to model multiple nodes. The modular approach is also used for message generators, i.e. message formatting.

Actual data to be transferred is not modeled. It is possible to abstract the real data from the model since, in this case, the data is not used for any sort of control of the system. A model of a higher level protocol based on CAN, e.g. CANopen, would require the actual data. The final CANopen model would look a bit different, but it could be based on the CAN model presented in this paper.

Since time is used for modeling, it is necessary to define how much real time is represented in a single simulation time unit. In this paper the single simulation time unit represents 10 ns of real time.

The top level of the model, given in Fig. 6, contains: two nodes (substitution transitions *Node_1* and *Node_2*); three places (*Node2Bus*, *BusFree* and

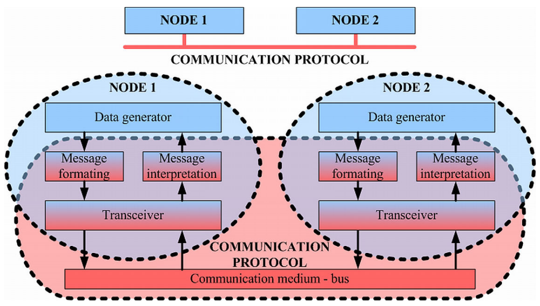

Fig. 5. The concept of the bus communication systems.

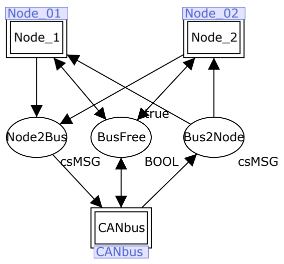

Fig. 6. The top level of the CAN bus communication model.

Bus2Node) that represent the state of the CAN bus; and a substitution transition (*CANbus*) that handles messages on the CAN bus.

The first node is configured as given in Fig. 7. The node sends 9 messages. All messages are to be sent with a 10 ms period. The length of messages is defined by the parameter DLC (number of data bytes), while ID defines message identifier.

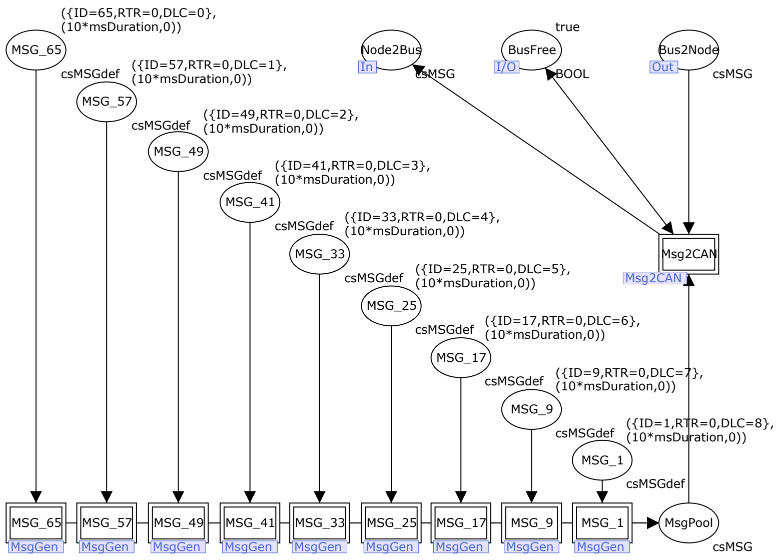
({ID=65,RTR=0,DLC=0}, (10*msDuration,0))
MSG_65
csMSGdef
({ID=57,RTR=0,DLC=1}, (10*msDuration,0))
MSG_57
csMSGdef
({ID=49,RTR=0,DLC=2}, (10*msDuration,0))
MSG_49
csMSGdef
({ID=41,RTR=0,DLC=3}, (10*msDuration,0))
MSG_41
csMSGdef
({ID=33,RTR=0,DLC=4}, (10*msDuration,0))
MSG_33
csMSGdef
({ID=25,RTR=0,DLC=5}, (10*msDuration,0))
MSG_25
csMSGdef
({ID=17,RTR=0,DLC=6}, (10*msDuration,0))
MSG_17
csMSGdef
({ID=9,RTR=0,DLC=7}, (10*msDuration,0))
MSG_9
csMSGdef
({ID=1,RTR=0,DLC=8}, (10*msDuration,0))
MSG_1
csMSGdef
Node2Bus
In
csMSG
true
BusFree
I/O
BOOL
Bus2Node
Out
csMSG
Msg2CAN
Msg2CAN
MSG_65
MSG_57
MSG_49
MSG_41
MSG_33
MSG_25
MSG_17
MSG_9
MSG_1
MsgGen
MsgGen
MsgGen
MsgGen
MsgGen
MsgGen
MsgGen
MsgGen
MsgGen
MsgPool
csMSG

Fig. 7. The CAN node, substitution transition label *Node_01*, Fig. 6.

The second node is configured as given in Fig. 8. The node sends 9 messages. All messages are to be sent with a 10 ms period. The length of messages is defined by the parameter DLC (number of data bytes), while ID defines message identifier.

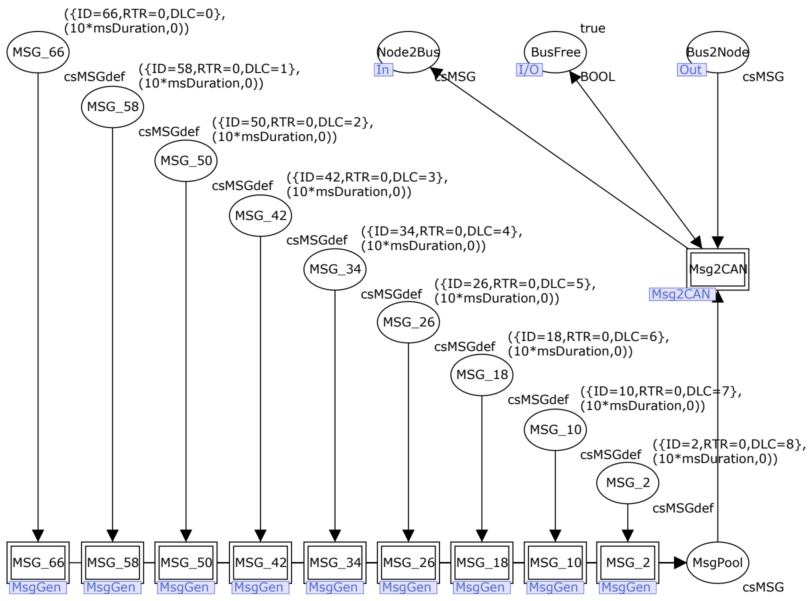

Fig. 8. The CAN node, substitution transition label *Node_02*, Fig. 6.

Each node uses nine message generators, modelled by substitution transition *MsgGen* in Fig. 7 and Fig. 8. Besides message generators, each node uses a substitution transition that forwards messages from the node to the CAN bus (substitution transition label *Msg2CAN*).

The message generator, given in Fig. 9, creates a single shot or periodic messages for the node to transfer. It is important to be able to create both types of messages, since start-up procedures and similar actions use one-time messages, while the system in operation usually uses periodic messages. Alarms or error situations in the system can be communicated by the one-time messages too.

Two functions are present in the arc inscriptions originating from the transition *ReadMsg*, Fig. 9. The first one is *abs(int)*. This built-in function returns the absolute value of an integer *int*. The second function is *fGenFrame(ID, RTR, DLC)*. This is a user defined function and it creates a CAN message frame based on the identifier (*ID*), type of frame (data or remote, *RTR*), and number of data bytes (*DLC*).

The system used to transfer generated messages to the CAN bus is given in Fig. 10. It is composed of two substitution transitions. The first one, *MSG2RAM*, takes generated messages and places them in the input FIFO memory buffer, just like a regular communication processor. The second one, *Transceiver*, takes

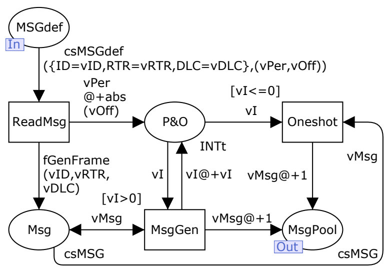

Fig. 9. The CAN message generator, substitution transition label *MsgGen*, Fig. 7 and Fig. 8.

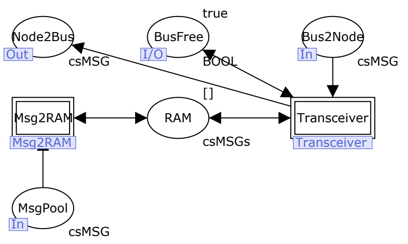

Fig. 10. The system used to transfer messages from the node to the CAN bus, substitution transition label *Msg2CAN.*, Fig. 7 and Fig. 8.

the first message from the input FIFO buffer and sends it over the CAN bus. *MSG2CAN* is structured that way in order to enable easier simulation of different message sorting mechanisms.

In *MSG2RAM*, simultaneously generated messages, e.g. the messages generated in the same interrupt routine, are sorted according to their priority, Fig. 11. The highest priority message (lowest ID) is on the top of the list, while the lowest priority message (highest ID) is on the bottom of the list. The entire sorted list is appended at the end of the input FIFO memory buffer. Thus the input FIFO buffer is not sorted, i.e. it can happen that a higher priority message is behind a lower priority message. The real system behavior is simulated that way. Other system behaviors can be simulated too, e.g. a system with a priority sorted input FIFO buffer.

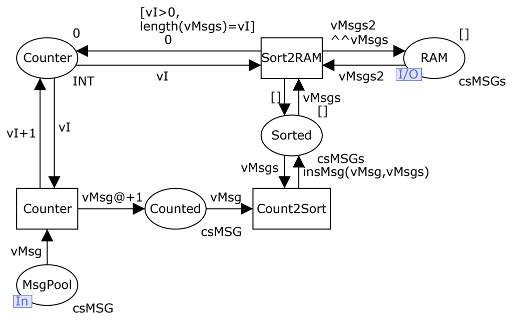

Fig. 11. The system used to transfer messages from RAM to the CAN transceiver, substitution transition label *Msg2RAM*, Fig. 10.

Substitution transition *MSG2RAM* has two functions on the arc inscriptions originating from transitions *Sort2RAM* and *Count2Sort*, Fig. 11. The first one is a built-in function *length(lst)*. It returns the length of the list *lst*, i.e. the number of elements on the list. The second function is a user defined *insMsg(msg, lst)*. This function inserts a message *msg* to the priority sorted list of messages *lst*.

Figure 12 shows the message transceiver. It takes only one message at a time. The message is taken from the top of the input FIFO buffer. The transceiver sends the message to the CAN bus (place *Node2Bus*) only if the bus is available. The state of the CAN bus is defined in the place *BusFree*. The CAN bus can be either available (*true*) or occupied (*false*).

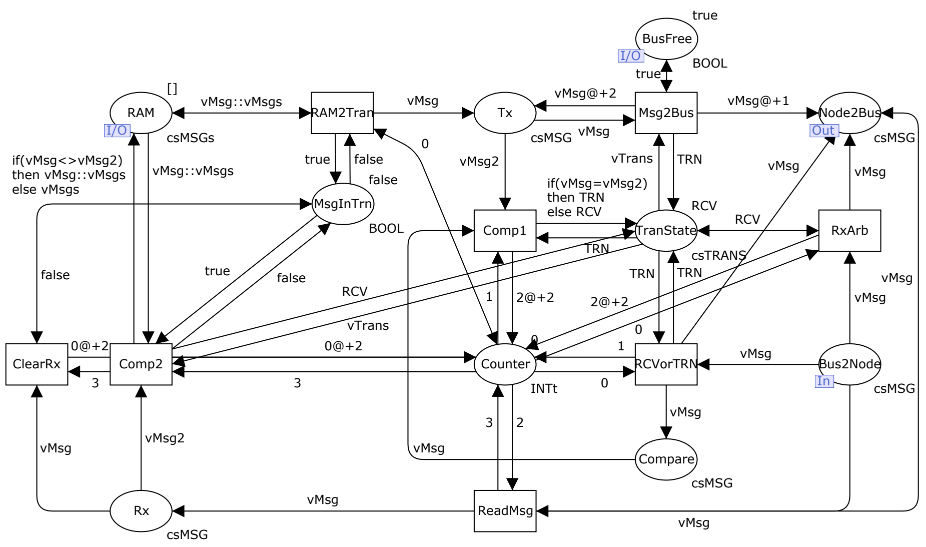

Fig. 12. The model of the CAN transceiver, substitution transition label *Transceiver*, Fig. 10.

When the node gains access to the bus, it sends the message from the transmit register (place *Tx*). If the message has the highest priority, compared to all the other messages sent by competing nodes, then the transceiver remains a transmitter. If the message has a lower priority, then the transceiver turns into a receiver. The message is handled by the *CANbus* module and the end result is passed to transceivers through the place *Bus2Node*.

The *CANbus*, given in Fig. 13 handles messages sent by the nodes. The maximum number of messages in the place *Node2Bus* must be equal or smaller than the number of connected nodes. Once the messages are received, the bus changes the state of the *BusFree* place. Next, the messages are sorted according to their priority. The function *insMsg(msg, lst)* adds message *msg* to the sorted list *lst*.

The highest priority message is sent to the place *Bus2Node*, Fig. 13. This way all nodes become aware of the highest priority message and can change

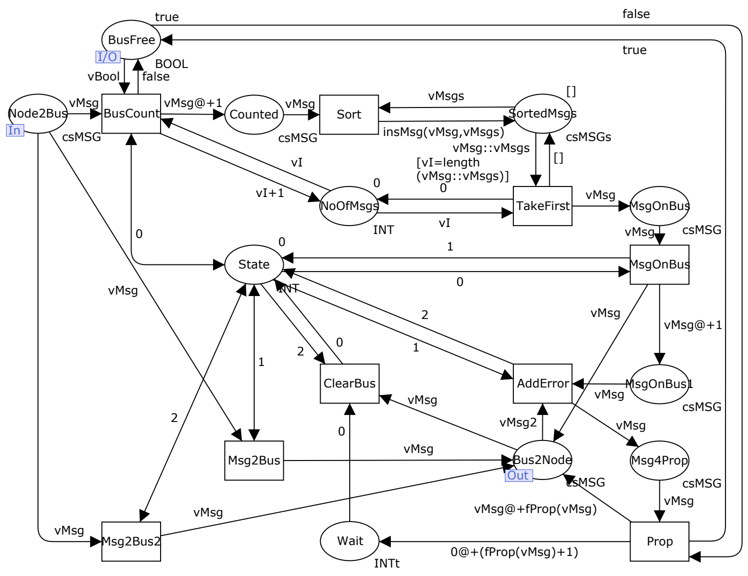

Fig. 13. The model of the CAN bus, substitution transition label *CANbus*, Fig. 6.

their respective state (transmitter or receiver). In order to synchronize the state change of all the nodes it is necessary to add a time delay on the arc between the transition *MsgOnBus* and the place *MsgOnBus1*. The message is then processed by the *AddError* transition. Here complex error behavior of the bus can be modeled. The message, modified or not, is sent to the place *Msg4Prop*. In this paper no errors on the bus were assumed.

Message propagation time is calculated by the transition *Prop*, Fig. 13. Depending on the size of the message, and other parameters influenced by the *AddError* transition, it is calculated how long the message actually occupied the bus. The message is sent to the *Bus2Node* place, where it is read by all the nodes. If there was an error during the transmission, the transmitting node would automatically try to retransmit the message.

The substitution transition *CANbus* uses two user defined functions, Fig. 13. First one is *insMsg(msg, lst)*. This function inserts a message to the priority sorted list. The second function, *fProp(msg)*, is used to calculate the message propagation time. It uses total number of bits in the message frame to calculate the duration on the CAN bus. All delays added in the *CANbus* are compensated for.

The place *State* in Fig. 13 defines the state of the CAN bus as follows: *0* - no message present on the bus; *1* - a single message present on the bus, which is used to inform the nodes about the message that won the arbitration process; *2* - a single message present on the bus which is used to transfer the message to all the nodes.

4.1 Model restrictions

The state machine of the CAN node, that depends on the error counters, was not implemented. Modeling the error counter would be a major difficulty for the verification. TRN - transmit error counter has 256 states. RCV - receive error counter has 128 states. This alone gives $128 * 256 = 32768$ states per node.

If there are occasional errors in the communication system, and we model them, then the model will behave in an identical way as the real-life system. In the case of a heavily disturbed communication (system error) the CAN nodes will eventually go into "bus off" state, [7, 8]. Heavy disturbances are usually caused by physical defects, i.e. short-circuit, wave reflection due to poor termination, disconnected wires etc. This is out of scope of our model.

5 Verification and validation results

5.1 Verification

Verification is used to test the model for desired (or undesired) properties, [18–20]. Since verification of a large system tends to get extremely difficult (due to the state explosion problem), it is possible to use modular analysis of the system, [13–17]. There are different possible approaches to the modular analysis.

It is possible to analyze every module as a separate entity. The boundary conditions should, in such cases, be identical to the ones when the module is part of the system. It is possible to define such conditions and mimic them. Thus, all possible local states of the module are checked.

Three types of modules can be used in a modular analysis approach: a source module, a transport module, and a sink module. A source module generates tokens without any external influence. A transport module transfers the tokens from the input place(s) to the output place(s). It does not generate any tokens on its own. The sink module consumes all tokens it receives from the input(s). The most critical modules are the transport modules. It is necessary for these modules to have independent input interfaces, i.e. if more than one input is present, then the inputs have to cause independent actions within the module.

There are three separate modules for the state space analysis in this paper: (i) message generator - generates messages for the node to send to the CAN bus, (ii) the node - gets messages from message generators and sends them via the CAN bus, and (iii) CAN bus - handles messages that the nodes want to send. In this paper message generator is the source module, CAN bus is the transport module, while the node is the sink module. For the state space analysis of all modules, the following branching condition has been used:

```
fn (n:Node) =>
    if (n=hd(sort INT.lt (EqualsUntimed(n))))
    then
        true
    else
        false
```

This condition limits the generation of the state space. States (nodes in the state space) are compared to each other, but without time marks. If there are two identical states with different time marks, only one will be processed. This is a perfectly legal condition since there are no time controlled events except the generation of the periodic messages. Properties of these periodic messages do not change with time. This means that a message generated at time $T1$ will cause identical system behavior as a message generated at time $T2$, where $T1 \neq T2$, if the state of the system is otherwise identical (time invariant system).

Message generator module. The input place to message generator module is *MSGdef*. The output place is *MsgPool*. The module has no other interaction with its environment, Fig. 14. The CAN message generator can generate two types of messages: (i) single message and (ii) periodically repeating message. The module has to be verified for both types.

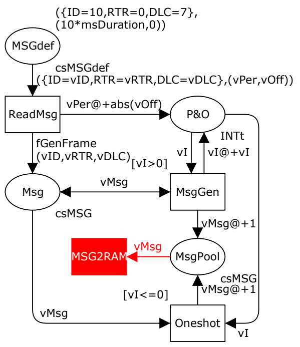

Fig. 14. The model used for verification of the message generator module. Filled transition is used to mimic the behavior of the environment.

The single message generator should have the following properties: (i) only one type of message is generated and (ii) if the module deadlocks, it terminates properly. Both properties have been verified by the state space analysis. The state space has 4 nodes and 3 arcs. There is one dead marking and there is one dead transition *MsgGen* (this transition creates periodic messages).

The periodic message generator should have the following properties: (i) only one type of message is generated, (ii) module does not deadlock, and (iii) module is in livelock. All properties have been verified by the state space analysis. State space has 4 nodes and 3 arcs. It is not the full state space because time changes, so no markings are identical. There is no dead marking and there is one dead transition *Oneshot* (this transition creates the single message).

Node module. The input places to the node module are *Node_1*, *BusFree* and *Bus2Node*. The output places are *Node2Bus* and *BusFree*. The module has no other interaction with its environment, Fig. 15. The CAN node can get two types of messages from the message generator: (i) single message and (ii) periodically repeating message. The module has to be verified for both types of messages.

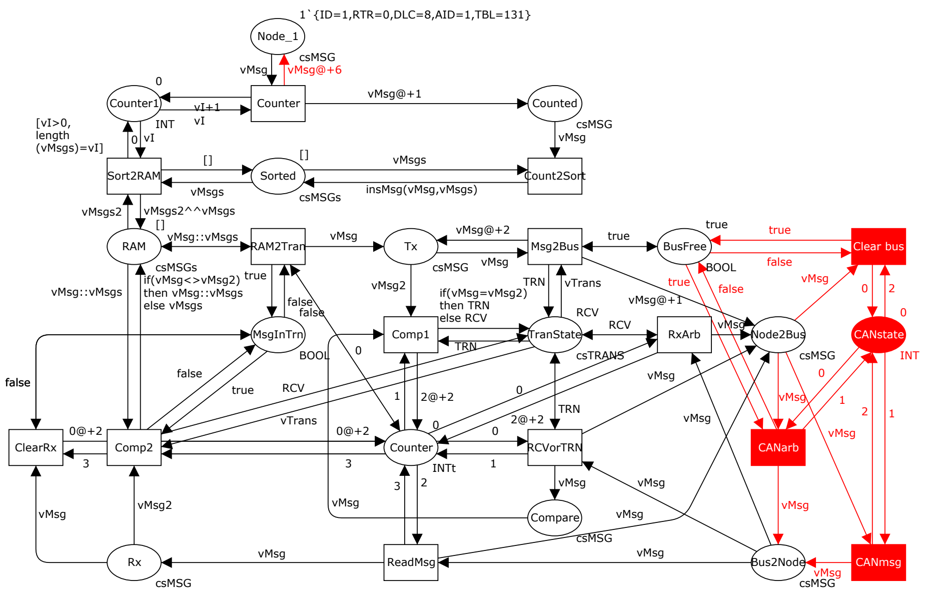

Fig. 15. The model used for verification of the node module. Filled places and transitions are used to mimic the behavior of the environment.

With the single message the node should have the following properties: (i) only one type of message is handled and (ii) module deadlocks. Both properties have been verified by the state space analysis. The state space has 14 nodes and 14 arcs. There is one dead marking and there are two dead transitions *ClearRx* and *RxArb* (these transitions are used when the node is receiver). The correct behavior of two nodes, each sending a single message (with different IDs), has also been tested. In such a case there is no dead transition. This test also verified the automatic retransmission capability.

With the periodic message the node should have the following properties: (i) only one type of message handled, (ii) module does not deadlock, (iii) module is in livelock, and (iv) only one message at the time sent to the bus. All properties have been verified by the state space analysis. The state space has 14 nodes and 14 arcs. It is not full state space because time changes so no markings are identical. There is no dead marking and there are two dead transitions *ClearRx* and *RxArb*. The correct behavior of two nodes, each sending a periodic message (with different IDs), has also been tested. In such a case there is no dead transition. This test also verified the automatic retransmission capability.

Creation of multiple messages and handling of these messages has only been simulated. There is no need for the verification of this property because messages with periods lower and higher than message propagation time have been generated for the state space analysis.

In the case of the periods lower than message propagation time (input arc to the place *Node_1*, Fig. 15), messages stack up in the node but the node sends only one message at a time. The state space analysis run terminates when the deadline, i.e. threshold time, is reached. In Fig. 15 the initial value of the place *Node_1* is set to the maximum message length, and a message period significantly lower than the propagation time. In the case of the period higher or equal to the message propagation time, all messages are sent and the buffer is empty. The analysis terminates properly, i.e. as expected.

CAN bus module. The input places to the CAN bus module are *BusFree* and *Node2Bus*. The output places are *Bus2Node* and *BusFree*. The module has no other interaction with its environment, Fig. 16. The CAN bus module can get two types of messages from the node: (i) a single message and (ii) a periodically repeating message. For the verification of the module this is not important. It is important that, while one message is being processed, another one does not access the bus. CAN bus was modeled with 5 different non-periodic messages, representing 5 different nodes.

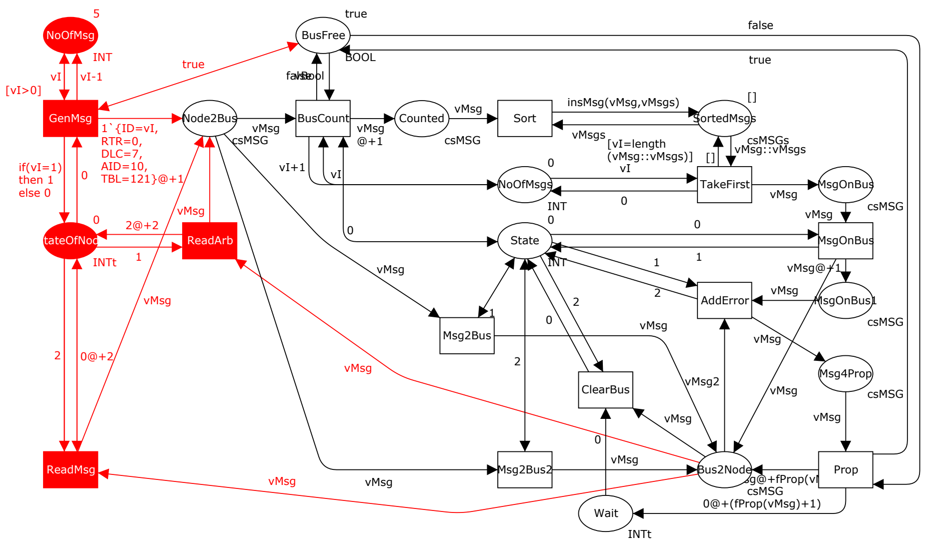

Fig. 16. The model used for verification of the CAN bus module. Filled places and transitions are used to mimic the behavior of the environment.

The CAN bus module should have the following properties: (i) multiple messages can access the *Node2Bus* place (max. number of messages equals max. number of connected nodes), (ii) only one message should be handled, (iii) message of highest priority should be handled, and (iv) deadlock can occur only in case there are no more messages to handle. All properties have been verified by the state space analysis. State space has 77 nodes and 174 arcs. There is one dead marking and no dead transition. *Node2Bus* holds at most 5 messages. Place *Bus2Node* contains the message of highest priority.

5.2 Validation

The CAN communication system consists of two nodes with the following properties: bit rate of 500 kbit/s, 11-bit message ID, each node sends 9 different messages (18 messages in total), all messages have different IDs, message lengths are in range from 0 to 8 bytes, no errors on the bus. The only thing to consider when developing the test system is that the propagation time of all messages has to be lower than half the round period. In this case total propagation time is cca. 3 ms while the round period is 10 ms, i.e. 3 ms $<$ 5 ms.

Two tests were conducted. The first test, Test1, considered messages with data that produced the highest number of bits (due to bit stuffing). The second test, Test2, considered messages with data that produced the lowest number of bits. Nodes were programmed as shown in Tab. 1 and Tab. 2.

Table 1. Message settings for two tests of Node 1.

Node 1			
ID	DLC	Test1	Test2
1	8	0x3C	0xAA
9	7	0xC3	0xAA
17	6	0x0F	0xAA
25	5	0xF0	0xAA
33	4	0x1E	0xAA
41	3	0xE1	0xAA
49	2	0xF0	0xAA
57	1	0xE1	0xAA
65	0	no data	no data

Table 2. Message settings for two tests of Node 2.

Node 2			
ID	DLC	Test1	Test2
2	8	0x3C	0xAA
10	7	0xC3	0xAA
18	6	0x0F	0xAA
26	5	0xF0	0xAA
34	4	0x1E	0xAA
42	3	0xE1	0xAA
50	2	0xF0	0xAA
58	1	0xE1	0xAA
66	0	no data	no data

Both nodes had to start sending messages at the same time. All messages are sent only once. The main reason is that the simulation model has the knowledge of global time, the simulation time, while the real-life equipment does not.

Real-life equipment has inherent drifts and needs synchronization in order to have a notion of global time. This was not implemented in the real-life system, so only one round of messages was allowed. The authors did actually test the real-life system allowing it to continue operation without synchronization.

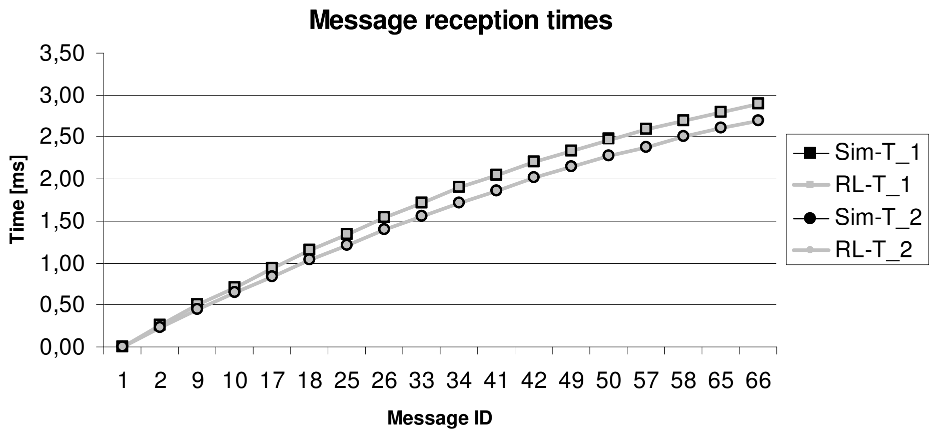

Fig. 17. The results of the validation tests. *Sim-T_x* stands for simulation test x. *RL-T_x* stands for real-life test x.

An internal test report [21] shows a significant drift (up to 10.5 ms per hour) between two identical oscillators. The same oscillators were used in the microelectronic boards for this paper. The 10.5 ms per hour drift translates to $0.0105/(60 * 60) = 2.917 * 10^{-6} = 2.917$ ppm.

Since a single bit lasts $1/500000 = 2 * 10^{-6}$ sec it can be seen that an offset of a single bit between the two boards will be within $2 * 10^{-6}/2.917 * 10^{-6} = 0.686$ sec. This is roughly 68 ms. This gives about 6 rounds of messages, at 10 ms period, without distortion. This calculation does not include other factors, such as interrupt latencies, which can cause drifting and significantly lower the number of synchronized rounds.

The difference was observed, in average, after the third round. Then a message sent by one of the nodes was sent later then planed, but still within the block of messages. After about 1-2 min the messages form the nodes were not interleaved any more but rather separate blocks of messages were observed on the bus. These results were much better at a lower communication speed (100 kbit/s) and with a lower number of messages (3 per node). Now the drifting became apparent after about 13 rounds (cca. 1.3 sec). The separate blocks of messages were observed after 7 min.

The real-life system had a starting message. It was not considered to be the starting point of time measurement, because of the message processing overhead, which is not modeled. The moment the first message (ID 1) was received, is considered the starting point, i.e. time 0. All other messages were timed according to the first message. The obtained results are given in Fig. 17.

The time difference, between the model and the real-life system, was in both tests less than 1 μs, so it is not visible in Fig. 17.

6 Conclusion and future work

It has been shown that the proposed concept for communication system modeling is adequate. The verification and validation results show that the simulation model can be used during the development of new CAN based systems. In future work a model of 16 (+4 optional) nodes will be created. This model will be used to simulate and analyze the CAN bus system in a future city train. Due to the high number of nodes, and even higher number of messages, a modular verification approach is necessary. Future work will address a more formal description of necessary and sufficient module boundary conditions.

In the future, gateway systems, based on real-life equipment, between the CAN and the WTB buses will also be developed.

Acknowledgments. This work was supported by Končar - Electrical Engineering Institute, Croatia, and the Ministry of Science, Education and Sports of the Republic of Croatia.

References

1. Marijan, S.: Control Electronics of TMK2200 Type Tramcar for the City of Zagreb. International Symposium on Industrial Electronics - ISIE, volume IV, Dubrovnik (2005) 1617–1622
2. Marijan, S.: Vehicle control unit for the light rail applications. International Conference on Electrical Drives and Power Electronics - EDPE, Dubrovnik (2005)
3. Bago, M., Marijan, S., Perić, N.: Modeling Controller Area Network Communication. International Conference on Industrial Informatics - INDIN, Vienna (2007) 485–490
4. Bago, M., Perić, N., Marijan, S.: Modeling Wire Train Bus Communication Using Timed Colored Petri Nets. International Conference on Instrumentation, Control and Information Technology - SICE, Tokyo (2008) 2905–2910
5. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software, vol. 20, No. 5 (2003) 19–25
6. Desel, J., Juhas, G.: What is a Petri Net?. Unifying Petri Nets, Springer-Verlag (2001) 1–25
7. ISO: Road vehicles - Controller area network (CAN) - Part 1: Data link layer and physical signalling. ISO 11898-1:2003, The International Organization for Standardization (2003)
8. ISO: Road vehicles - Controller area network (CAN) - Part 2: High-speed medium access unit. ISO 11898-2:2003, The International Organization for Standardization (2003)
9. Jensen, K.: An Introduction to the Theoretical Aspects of Coloured Petri Nets. A Decade of Concurrency, Lecture Notes in Computer Science, vol. 803, Springer-Verlag (1989) 230–272
10. Jensen, K., Kristensen, M.L., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer, vol. 9, No. 3-4 (2007) 213–254
11. Standard ML of New Jersey. http://www.smlnj.org

12. Ullman, J.D.: Elements of ML Programming. Prentice-Hall, Englewood Cliffs (1998)
13. Valmari, A.: The State Explosion Problem. Lectures on Petri Nets I: Basic Models, Lecture Notes in Computer Science, vol. 1491, Springer-Verlag (1998) 429–528
14. Valmari, A.: State of the Art Report: STUBBORN SETS. Petri Net Newsletter, No. 46 (1994) 6–14
15. Christensen, S., Petrucci, L.: Modular Analysis of Petri Nets. The Computer Journal, vol. 43, No. 3 (2000) 224–242
16. Lakos, C., Petrucci, L.: Modular Analysis of Systems Composed of Semiautonomous Subsystems. Fourth International Conference on Application of Concurrency to System Design - ACSD, (2004) 185–195
17. Lakos, C., Petrucci, L.: Modular state space exploration for timed petri nets. International Journal on Software Tools for Technology Transfer, vol. 9, No. 3-4 (2007) 393–411
18. Toussaint, J., Philippe, C., Simonot-Lion, F.: A Model of CAN-based Applications for the Verification of Temporal Properties. 3rd IFAC Symposium on Intelligent Components and Instruments for Control Applications - SICICA, Annecy, (1997)
19. Krakora, J., Hanzalek, Z.: Timed Automata Approach to CAN Verification. 11th IFAC Symposium on Information Control Problems in Manufacturing - INCOM, vol. 1 (2004)
20. Liu, L., Billington, J.: Verification of the Capability Exchange Signalling protocol. International Journal on Software Tools for Technology Transfer, vol. 9, No. 3-4 (2007) 305–326
21. Bago, M.: Test report - oscillator drift. Končar - Electrical Engineering Institute (2005) 1–12