UML-based framework for simulation of distributed ECU systems in automotive applications

Frank Steinert • proTime GmbH • Prien • Germany

A UML based framework for the simulation of distributed systems of embedded control units (ECUs) has been developed by proTime. This framework generates at its runtime an executable model of a system, described by an editable system description. Thereby this simulated system can be used for tests and analysis.

A basis for this UML based framework are executable models of different field busses. Currently models exists for CAN and FlexRay, Wakeup Line and a real-time control bus for simulation control purposes. These field bus models will be used (even simultaneously) to connect the modeled ECUs while the modeled system is under simulation.

This UML based framework allows both the simulation of communication processes, and the simulation of functionality depending on these communication processes all with the correct timing.

The main advantage of this UML based framework is the testability of communication dependant functionality like gateways, control loops and network management without an expensive test environment.

The principal theme of this speech will be the CAN bus model, its integration into the runtime-created system model and the resulting (almost boundless) possibilities. Currently automotive systems are the focus of our modeling and simulation, but the applicability of this UML based framework is not restricted to these tasks.

1. The CAN Bus Model

1.1 The Basics

The occasion for the development of the UML based CAN bus model was a concrete problem from the domain of board net communication in automobiles. There was a matter of coupling the CAN based protocols for the network management of two communication nets by using a gateway. There were already drafts for the functionality of such a coupling, but the different operating conditions under which the system should operate reliably, were not to be completely grasped analytically. A realization in hardware seemed too luxurious, which is why the new way of a modeling and simulation was considered. The modeling should be implemented using UML. For various reasons Rhapsody by I-Logix (see www.ilogix.com) was selected which comes with an excellent code generator and allows the easy execution of the models.

One can summarize the essential requirements of the CAN bus model as follows:

- The transmission of the CAN telegrams on the CAN bus should be able to be simulated as far as possible with chronological correctness. Besides, the main focus does not consciously become the chronologically correct mapping of the bus states (sample points, bit times etc.).
- The arbitration is to be modeled correctly.

- The CAN bus model can be simply reconfigured for the simulation of different scenarios
- The CAN bus model is to be able to simulate several instances of the model at the same time.
- A model of a CAN controller is to be provided which allows the very simple binding of existing driver implementations. This model must be able to be instantiated several times at the same time.

The CAN bus model simulates the CAN bus in terms of functionality, so that the modeling of the CAN bus can be abstracted from many of its aspects.

1.2 Modeling Of Time

An essential problem of the behavioral modeling is the mapping of the timeline. Taking into account the purpose for which the model should be developed the following decisions were met:

- The time behavior of the modeled CAN bus triggers all the other functions of the model (e.g., the modeled CAN controllers).
- Only the transmission of telegrams (of their bits) on a modeled CAN bus "uses" time. All the other functions operate during simulation with infinite performance.

Therefore the time is measured with a resolution which corresponds to the bit time on a CAN bus. In each case at the end of such a time step the time is stopped for so long as needed, until all actions pending at this time step are carried out. Only when there is nothing more at this time step to do (or to be simulated), the time advances one step.

1.3 CAN Bus and CAN Controller

Because of the realization of the described time model the CAN Bus model is reducible to provide a time base triggering the model's behavior and to transport CAN telegrams between the simulated CAN controller's models.

Taking into account the objective for which the model should be developed the following decisions were met:

- Bit stuffing is not taken into consideration, however, it can be upgraded any time without much effort.
- Error frames are not generated, because no errors can be expected from a undisturbed CAN bus. However, it is possible without much effort, to model the appearance of bit errors and the generation of error frames.
- The acknowledge bit is modeled indirectly: so long as only one controller is on the bus, its upload is not concluded (no "transmit interrupt" is triggered).
- The CAN controllers are very simply realized: Basic CAN without acceptance filtering. Because of the "infinite" performance style of simulation, this is not a restriction.
- The interrupts of a CAN controller are mapped using signals (events) which control the state machines of the model.
- The configuration of the CAN bus and the CAN controllers is read in by the simulation, at start up, from a file. This file is easily modified with a simple text editor.

In contrast to the reality where a CAN bus with its 2 wires has no behavior of its own (at least at the level of abstraction of interest to us) a part of the behavior of the CAN controllers is realized by the CAN bus model. But also in reality the CAN bus without its CAN controllers would be not really a CAN bus! For modeling reasons it is acceptable to give behavior to the CAN bus - the synchronization of the modeled CAN bus controllers is to be realized substantially easier over an active bus at the telegram level, because here the correct bit timing does not need to be modeled.

In Figure 1 the state chart of a simplified CAN bus state machine is illustrated which models the relevant behavior of a CAN bus.

The method "GetNextMessage" (transition from "ST_Empty" to "ST_InTrasmition") checks all CAN controllers registered with the CAN bus for the CAN telegram to be sent with the highest priority and fetches it. The machine remains in the state

"ST_InTrasmition" for long as the duration of transmission of this CAN telegram. The duration of the telegram transmission depends on the number of data bytes (the Stuff Bits are ignored) in this telegram. With the method "SendNextMessage" all controllers are informed about the completed transmission of a telegram on the CAN bus and they receive this telegram. The sending controller interprets this information as a transmission confirmation. As a result the bus is blocked to model (c_blockTicks) the INTERFRAME SPACE and afterwards the next telegram, which is already waiting, will be sent. If no telegram is to be dispatched, it waits for one. With "evCANSendMessage" it is signaled that a telegram was handed over for sending.

The lower partial machine provides information about the model regarding whether bus traffic actually occurs or not. This can be used for test purposes.

OVERLOAD- and ERROR-Frames are not modeled. However, these could be interesting for the simulation of error scenarios and could be inserted with low effort into the model.

The modeling of REMOTE-Frames is a task of the CAN controller's model.

The INTERFRAME SPACE is modeled without "Suspend Transmission" - but this too can be inserted into the model with minimal effort. For our previous work these unmodeled parts of the specification were not important.

The application interface of the CAN bus model is provided by an abstract interface class which is implemented by the CAN bus model. Figure 2 shows this interface class. The names of the methods are generally self explanatory, with an exception: the method "register controller" serves to announce the (simulated) CAN controllers to the bus model - only the CAN controllers known to the bus model can send and receive CAN telegrams using this bus model. Of course it is possible to generate any number of CAN busses and CAN controllers in any configuration and allocation at runtime. The generation of CAN busses and CAN controllers is done by (likewise modeled) a builder at runtime which takes its information from an editable file.

The interface of the CAN bus controller is likewise provided by an abstract interface class. This interface is implemented by the CAN controller model. In Figure 4 you can see the relevant class. Also here the names of the methods are generally self explanatory. Beyond that, the class has further methods which are used by the CAN bus model to inform it about the sending or the receipt of CAN telegrams. In principle it is possible on account of the modularity of the model to also create additional controller models and to insert them into the model. These could then be operating in the same simulation beside the other controller models in parallel.

2. Additional Bus Models

Beside the described CAN bus model models were already realized for FlexRay, the wake up line and a real-time control bus. The real-time control bus manages the simulation. The wakeup line models a digital signal for waking up (simulated) ECUs. For these bus models there are likewise suitable controller models. All bus models are naturally instantiable multiple times and are generated and linked by the builder during runtime according to a control file.

The models for the wake up line and the real time control bus are built up similarly to the model of the CAN bus. The relevant bus model has in each case a more complicated behavior than in reality. With the FlexRay model the division of tasks occurs between the controller and the bus in another way. Here the bus model serves merely as a clock generator, while it generates the so-called „macro ticks " and transmits them to all connected FlexRay controller models as trigger events for the state machines inside each controller model. The state machine's state chart of this clock generator is shown in illustration 3. The FlexRay controller model realizes here completely the winding up of the more complicated protocol and uses the FlexRay bus model merely for the transmission of the FlexRay frames. By contrast the bus model with the other modeled buses actively takes over the transmission of the telegrams and

realizes the respective transference protocol.

All bus models created up to now are several times instantiable and can be executed in the same simulation.

2.1. Extended Modeling Of Time

On account of the realization and integration of further bus models it was necessary to extend the time model. Thus the FlexRay works, e.g., with a time resolution in the ns range. In a system to be simulated the time model is driven accordingly by the bus model which works with the smallest "time quantities".

3. Modeling Of Bus Drivers

For the use of the bus models in a simulated environment (ECU), the bus drivers from which the bus model forms an accessible application package are required in implemented software. In our framework for this purpose abstract superclasses were defined which encapsulate the communication mechanisms of the bus models and make it accessible to a driver. In illustration 5 the class diagram of an example CAN driver is shown. In principle the models are so open that any drivers can be realized.

4. Simulation Framework

Building up on the model elements described up to now (bus models, bus controller model, driver models) an extensive simulation framework was constructed. This simulation framework enables the simulation of extensive distributed systems in their complexity.

4.1 Structure Element Node

The structure of a system to be simulated is based on the elements typically found in the automobile environment: bus, ECU and gateway. An ECU is a device communicating using one or several busses which executes application programs and accesses actors and sensors in the vehicle. A gateway provides communication between different busses and is a very complicated component, because it doesn’t only transmit simple telegrams from one to another bus. Furthermore, gateways can also be a component of an ECU. Beyond that, there are the higher communication protocols which are to be used by the ECUs. These higher communication protocols allow the transmission of bigger data volumes to control (transport protocols) or the system (network management, vehicle state management). Also there are the more or less standardized software layers which realize, e.g., the mapping of the signals generated or consumed by applications (e.g., a temperature, a command) on telegrams and frames.

The framework realizes these abilities of the ECUs by means of so called nodes. Besides, a node can contain one or several (different or of the same kind) bus bonds, a gateway and different applications. In the automobile sphere such a node always contains components of the vehicle state management and the network management, the latter also depending on the stamping of the respective bus.

4.2 Integration Of User Applications

An interface is provided for the simulation of applications. These applications can be executed within the systems provided with the framework. This interface is realized in form of an abstract superclass which contains all methods usable by an application. With these methods an application can send signals and receive. With these methods it is also controlled by the simulated vehicle state management. Among the rest, an element of this abstract superclass is an attribute which carries the name of an application. With this name an application is identifiable and can be associated, above all, in the system definition of one or several nodes.

4.3 Definition Of Systems To Be Simulated

To simplify the operation of the frameworks the framework reads in the structure information from a control file by the start of a simulation. This is possible, because behavioral model, behavioral implementing and the mapping of the system structure were decoupled from each other. By the realized decoupling of the system structure the simulation

framework gets practically usable and user-controllable. A builder integrated in the framework generates all of the required objects (instances of the bus models, Nodes etc.) at run time and links up them according to the control file. Besides, it is possible to generate this control file from other data sources, e.g., of a board net database. Beside the structure information the communication profiles (which telegrams or frames from a node will also get generated or consumed which signals are mapped like on telegrams and frames are taken ...) from the control files. Same applies to the parameterization of the gateway model. Also the gateway model needs information about which reception telegrams or frames are to be mapped in which way in transmit telegrams or frames. Figure 6 shows a sample of a simulation screen dump.

4.4 Sample Of Usage

With the help of the following example the use of this simulation-framework can be made clear by the hedging of functional specifications .

CAN based networks of ECUs in a car are coupled via gateways. Besides, the network management is realized on each of the nets according to OSEK-NM. The OSEK-NM provides for the fact that all ECUs are woken up if an ECU becomes awake and all ECUs in sync fall asleep when all ECUs are ready. With the coordination of the network management on the single buses by a real gateway false functions were observed in the practice over and over again. These led to the fact that the asleep-processes were delayed and states even appeared where only one part of the ECUs fell asleep and this was on account of the activity of the other ECUs which then woke again etc. The cause was discovered by analysis in the synchronization mechanism of the gateway. A new procedure was developed for this synchronization mechanism. By means of the simulation (about bus coupled simulated ECUs and gateways with OSEK-NM) it could be proved that this new procedure also operates under "adverse" edge conditions correctly. For this "just" the different test scenarios which create exactly these edge conditions were modeled and executed in the simulation execute. Besides, it was substantially easier to create complex test conditions with the model than in a real bus.

The evaluation of the NM specific behavior of the gateway and the nodes was developed by an additional node element which NM-Checker realizes. This NM-Checker observes the CAN telegrams communicated on the simulated busses and evaluates them. As soon as it ascertained a violation of the OSEK-NM specification, these were captured.

On account of the strict encapsulation of all model elements it was possible to extract afterwards the NM-Checker from the framework and to embed in a test environment. This test environment allows the access to real busses, the real systems in the car. Therefore it became possible, to certify the NM-Checker in the simulation, and also to evaluate the behavior to be of more real use in hardware with realized functionality.

Frank Steinert
proTime GmbH
J.-v.-Fraunhofer-Str. 9 • D-83209 Prien
+49(8051)6916-0
+49(8051)6916-11
Frank.Steinert@protime.de
http://www.protime.de

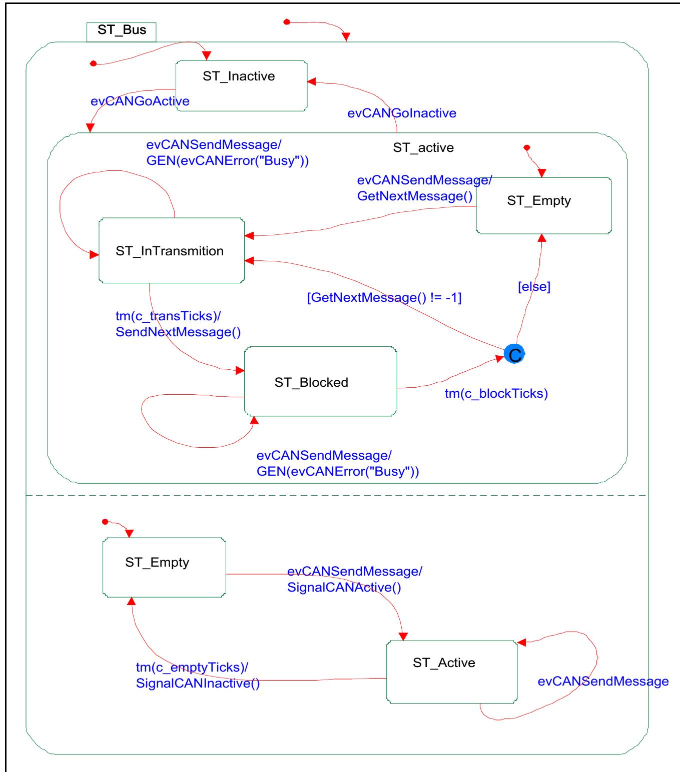

Figure 1: Statemachine of the CAN bus model

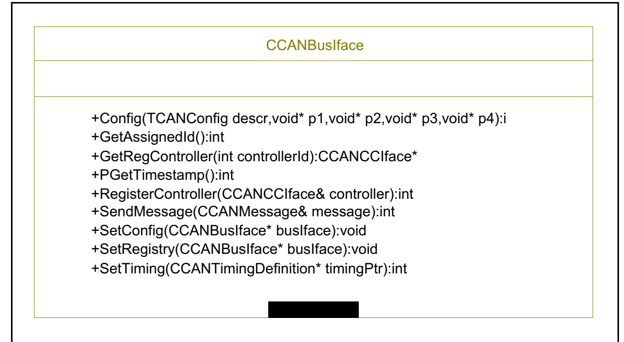

Figure 2: public interface of the CAN bus model

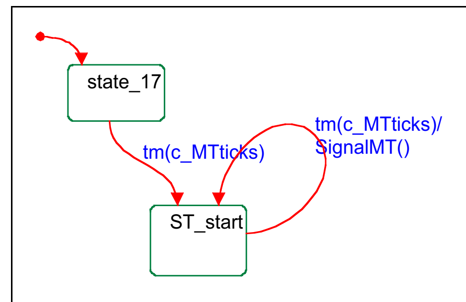

Figure 3: Statechart of the FlexRay model

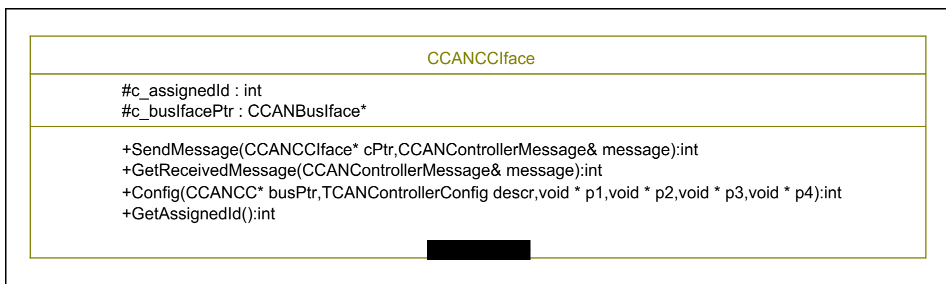

Figure 4: Public interface of the CAN controller

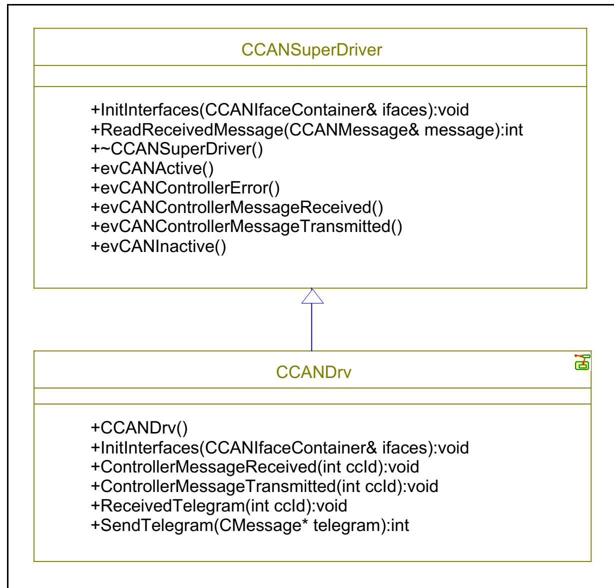

Figure 5: A sample for a CAN driver

C:\Programme\Rhapsody5.2\rhapsody.exe

```
node2:CAN1/CAN-Konfiguration in '\ARBEIT\CONFIG\pGWTable.ini'
Konfiguration der Nachricht 'TEL_CAN_IN_0001' eingelesen
Konfiguration der Nachricht 'TEL_CAN_IN_0002' eingelesen
Konfiguration der Nachricht 'TEL_CAN_IN_0003' eingelesen
Konfiguration der Nachricht 'TEL_CAN_IN_0004' eingelesen
Konfiguration der Nachricht 'TEL_CAN_OUT_0001' eingelesen
00:00:00,000.001  00:00,000.001    node2                                          Create FITTER: ( CAN1/CAN )
Die Nachrichten-Konfiguration - '\ARBEIT\CONFIG\FZMM_config.ini' eingelesen.
00:00:00,000.001  00:00,000.001    node2                                          Create FZMClient_ComManager: ( node2 )
00:00:00,000.001  00:00,000.001    node2                                          Create FZMClient_StateManager: ( node2 )
Die Nachrichten-Konfiguration - '\ARBEIT\CONFIG\pComTable.ini' eingelesen.
00:00:00,000.002  00:00,000.002    node2                                          Create COM: ( node2 )
00:00:00,000.002  00:00,000.002    node2                                          Create NM_BUSMANAGER: ( CAN1/CAN )
Die Timing-Konfiguration - '\ARBEIT\CONFIG\pNM_Config.ini' eigelesen.
Konfiguration der Nachricht 'NODE1' eingelesen
Konfiguration der Nachricht 'NODE2' eingelesen
Konfiguration der Nachricht 'NODE3' eingelesen

 Node:node3->enter state Bus Sleep
00:00:00,000.002  00:00,000.002    node3                                          Create ASCAN: ( CAN1/CAN )
node3:CAN1/CAN-Konfiguration in '\ARBEIT\CONFIG\pGWTable.ini'
Konfiguration der Nachricht 'TEL_CAN_IN_0001' eingelesen
Konfiguration der Nachricht 'TEL_CAN_IN_0002' eingelesen
Konfiguration der Nachricht 'TEL_CAN_IN_0003' eingelesen
Konfiguration der Nachricht 'TEL_CAN_IN_0004' eingelesen
Konfiguration der Nachricht 'TEL_CAN_OUT_0001' eingelesen
00:00:00,000.002  00:00,000.002    node3                                          Create FITTER: ( CAN1/CAN )
Die Nachrichten-Konfiguration - '\ARBEIT\CONFIG\FZMM_config.ini' eingelesen.
00:00:00,000.002  00:00,000.002    node3                                          Create FZMClient_ComManager: ( node3 )
00:00:00,000.002  00:00,000.002    node3                                          Create FZMClient_StateManager: ( node3 )
Die Nachrichten-Konfiguration - '\ARBEIT\CONFIG\pComTable.ini' eingelesen.
00:00:00,000.002  00:00,000.002    node3                                          Create COM: ( node3 )
00:00:00,000.002  00:00,000.002    node3                                          Create NM_BUSMANAGER: ( CAN1/CAN )

 Node: node1->enter state repeat Message

 Send NmMsg
00:00:01,823.474  00:01,823.474    node1:FITTER(CAN1/CAN)                         251 rec message     ( 08 ) fb 00 00 00 00 00 00 00
                             --- INFO --------------------------------------------  : Transmit to cc:100; Ergebnis: 1; erweiterter Fehlercode: 0
00:00:01,823.474  00:01,823.473    node1:FITTER(CAN1/CAN)                         251 send telegram   ( 08 ) fb 00 00 00 00 00 00 00
                             --- INFO --------------------------------------------  CAN-Bus ID:10
00:00:01,823.587  00:01,823.587      251 send message: ( 08 ) Data: fb 00 00 00 00 00 00 00
00:00:01,823.587                   CAN-bus ID:10                                  251 send telegram   ( 08 ) fb 00 00 00 00 00 00 00
00:00:01,823.587  00:01,823.586    node2:FITTER(CAN1/CAN)                         251 rec telegram    ( 08 ) fb 00 00 00 00 00 00 00
00:00:01,823.587  00:01,823.585    node3:FITTER(CAN1/CAN)                         251 rec telegram    ( 08 ) fb 00 00 00 00 00 00 00

 Send NmMsg
00:00:01,873.474  00:00,050.000    node1:FITTER(CAN1/CAN)                         251 rec message     ( 08 ) fb 00 00 00 00 00 00 00
                             --- INFO --------------------------------------------  : Transmit to cc:100; Ergebnis: 1; erweiterter Fehlercode: 0
00:00:01,873.474  00:00,050.000    node1:FITTER(CAN1/CAN)                         251 send telegram   ( 08 ) fb 00 00 00 00 00 00 00
                             --- INFO --------------------------------------------  CAN-Bus ID:10
00:00:01,873.587  00:00,050.000      251 send message: ( 08 ) Data: fb 00 00 00 00 00 00 00
00:00:01,873.587                   CAN-bus ID:10                                  251 send telegram   ( 08 ) fb 00 00 00 00 00 00 00
00:00:01,873.587  00:00,050.000    node2:FITTER(CAN1/CAN)                         251 rec telegram    ( 08 ) fb 00 00 00 00 00 00 00
00:00:01,873.587  00:00,050.000    node3:FITTER(CAN1/CAN)                         251 rec telegram    ( 08 ) fb 00 00 00 00 00 00 00

 Send NmMsg
00:00:01,923.474  00:00,050.000    node1:FITTER(CAN1/CAN)                         251 rec message     ( 08 ) fb 00 00 00 00 00 00 00
                             --- INFO --------------------------------------------  : Transmit to cc:100; Ergebnis: 1; erweiterter Fehlercode: 0
00:00:01,923.474  00:00,050.000    node1:FITTER(CAN1/CAN)                         251 send telegram   ( 08 ) fb 00 00 00 00 00 00 00
                             --- INFO --------------------------------------------  CAN-Bus ID:10
00:00:01,923.587  00:00,050.000      251 send message: ( 08 ) Data: fb 00 00 00 00 00 00 00
00:00:01,923.587                   CAN-bus ID:10                                  251 send telegram   ( 08 ) fb 00 00 00 00 00 00 00
00:00:01,923.587  00:00,050.000    node2:FITTER(CAN1/CAN)                         251 rec telegram    ( 08 ) fb 00 00 00 00 00 00 00
```

Figure 6: Screen dump of a simulation