Describing Communication Scenarios in Automotive Systems

Matthias Stümpfle, Markus Eberspächer

Institute of Communications Switching and Data Technics
University of Stuttgart
Seidenstraße 36, D-70174 Stuttgart
Phone: +49-711-121-2485 Fax: +49-711-121-2477
email: stuempfle@ind.e-technik.uni-stuttgart.de

Abstract

This paper gives an introduction into problems when specifying communication scenarios in automotive systems. Therefore communication requirements are stated and transferred to a common model. Derived from this a set of language elements is proposed to fulfill the descriptive needs. Finally a multitarget compiler and development environment is introduced to transform the described communication scenario into any desired hardware system.

1 Introduction

Automotive systems such as cars, vans, etc. are becoming increasingly complex. Especially the desire for more intelligent communication between electric and electronic devices, so called electronic control units (ECU), is steadily increasing. Cable lengths of up to two km for current standard solutions raise cost, vehicle weight and degrade its property for easy maintenance.

One first step to reduce cost and complexity in nowadays car communication systems is the use of serial buses to connect the ECUs. Using serial bus systems on the other hand leads to new problems in designing the electronic system of a vehicle; designers now must be able to construct well-designed network and layered software architectures. This is important particularly when products of different manufacturers have to work together in one net.

To ease the use of such a net appropriate software and tools must be made available to the users. The question that arises in this context is

How may such a complex system be described as easy as possible by e.g. hiding unnecessary implementation details of the system?

We found a two-way system to be most effective. First the architecture of the system has to be described. That means defining the ECUs in the net and the information that is transmitted over the net. One base for this description is the commonly used communication matrix. Secondly providing an application programming interface "API" to the programmer that makes it easy for him to use the communication facilities with some standardized functions that are always looking the same. This could be achieved by using object-oriented design methods and implementations.

This paper is concerned with an introduction into the problems of the specification of communication scenarios in automotive systems. Section 2 describes the needs and requirements for communication in such an environment. In Section 3 we classify these communication mechanisms and therefrom derive a language that fulfills the requirements (section 4). The result is shown in a short example in the appendix. Section 5 gives a brief introduction into the Open Tool Environment "ToolBench" in which the language and its compiler is integrated.

2 Requirements for a communication description

If describing communication scenarios the requirements and needs for such a description must be obtained. We support a customer-supplier view in this paper. The customer is the car manufacturer who has many suppliers that supply his product, the car, with several subproducts, forming a working network. From discussions with car producers and research in the common communication field [1] we found several major points that have to be taken into consideration when specifying a communication description or even a language. Among them are:

- the need for a standardized specification
- the need of respecting several rules that have to be obeyed in the development process
- the need for abstraction
- the need to describe all necessary communication types

The first three aspects will be discussed briefly in the following sections. The need to describe all necessary communication types will be explained in more detail in section 3.

Need for standardized specification

Due to different suppliers working on one network there has to be a general way to communicate with each other during the design and implementation process. A common language therefore must be provided to enable the customer of these suppliers to specify his needs. The client generates this specification with any kind of tool (text-editor, graphical user interface) and presents its description to the supplier who knows how to transform the specification into the desired product. In our case especially the needs for communication are considered.

Need of respecting the development process of a car network

In general the designing of an automotive network is a corporate work of different people in different companies. The designer of the car (in this case the customer) defines its requirements for the complete system. He plays the integration part in the development process [2]. Then he separates the functionality into several parts (ECUs) that have to be provided by the different suppliers. To perform the desired functionality the supplier does not need the full information about all communication relations going on in the network. Sometimes it's even desired that the supplier does not know all system details. He just needs to know what information he may rely on and which information he is supposed to supply.
This requirement demands a hierarchical description system with independent parts. One way is to provide „include"-possibilities. The designer of a supplier-company gets a file with all information entering and leaving his ECU. He includes this file into his own part of the description and will be able to produce the desired part of the net.

Need for abstraction

As already mentioned electronic systems are becoming more and more complex. In general this should impose no problem to the programmer of an application. His process running on an ECU does not necessarily have to know how the underlying hardware of the ECU is working. He simply has to know a set of operating system calls that can be used in his application. Information about the implementation of these calls can be kept away from him by using an

easy-to-use abstraction for the architectural aspects and an easy-to-use programming interface. Driver functionality for example may also be inserted by include-files written in the description language. These include-files will be provided by the designer of the driver who knows best its internal structures.

3 Classification of communication types

Certain functions of the electronic system of a vehicle are grouped into ECUs. These functions - implemented as processes running on processors or microcontrollers - either belong logically or topologically together. A process is defined as a logical sequence of statements to perform a certain functionality.
We now introduce two different views to describe communication between processes. They are different in the degree of abstraction. The first view, specifying the location of sender and receiver, is an abstraction level that can be directly provided by the application programmer. This is the kind of view the customer prefers. The only information he needs is which other processes and signals do exist in the system that want to communicate with him.
The second view is of general character and refers more to the internal structure of the communication soft- and hardware. This is the level that should be hidden to the user and therefore shall be automatically generated by appropriate tools.
To achieve this you will find that the second view may somehow be derived from the first view.

3.1 Producers and Consumers of Information

Inside ECUs and between them information must be shared and exchanged. Our goal is to achieve a maximum of performance and usability. From this point of view we distinguish among three kinds of communication types (see figure 1):

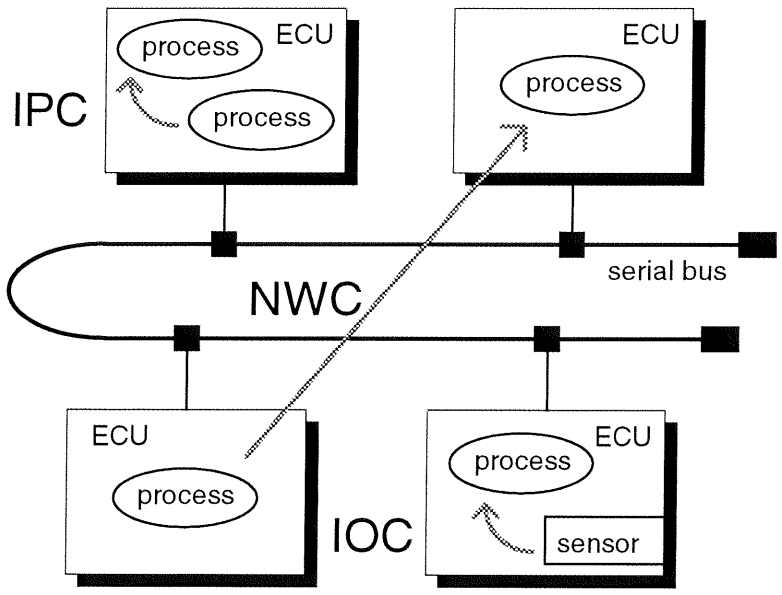

Figure 1: Types of communication

- Interprocess-Communication (IPC), which represents the exchange of information between processes that are located in the same ECU. Processes may either run on one processor or on several processors in the same ECU.

- Network-Communication (NWC) between related processes located in different ECUs using any sort of underlying communication network (i.e. CAN, VAN, SCP).

- Communication between processes and I/O-facilities (IOC), such as sensors and actuators.

This classification represents the view of an application programmer. He sees his ECU with some processes on it that produce and consume signals. So this is a very abstract view with very little information about the way (hardware!) the information has to pass to reach its destination.

3.2 General Communication Requirements

Besides the intuitive point-to-point view described in the previous section another more general view of communication paths has to be considered. Also 1:n or even m:n communication relations have to be provided by the system.
Consider e.g. a signal „EngineSpeed", i.e. an information unit carrying the information about the rotation speed of the engine. This signal is generated by one ECU placed near the engine and may be of interest to several other ECUs that need this information to calculate their results. Even processes of the same ECU may need this signal. Providing 1:n communication solves this problem: One supplier sends its signal to different consumers regardless the underlying hardware. (This scenario is shown in figure 2. Signal_3 is produced by process_C. It is sent to process_D and process_E on the same ECU and at the same time transmitted via the serial bus system to a receiver that is not part of this ECU).
The other necessity that has to be supported is a kind of m:n communication that forms part of the busses' capabilities. An example for an appropriate scenario is also shown in figure 2. Three processes (A-C) want to transmit their signals to processes, that are not located on their ECU.

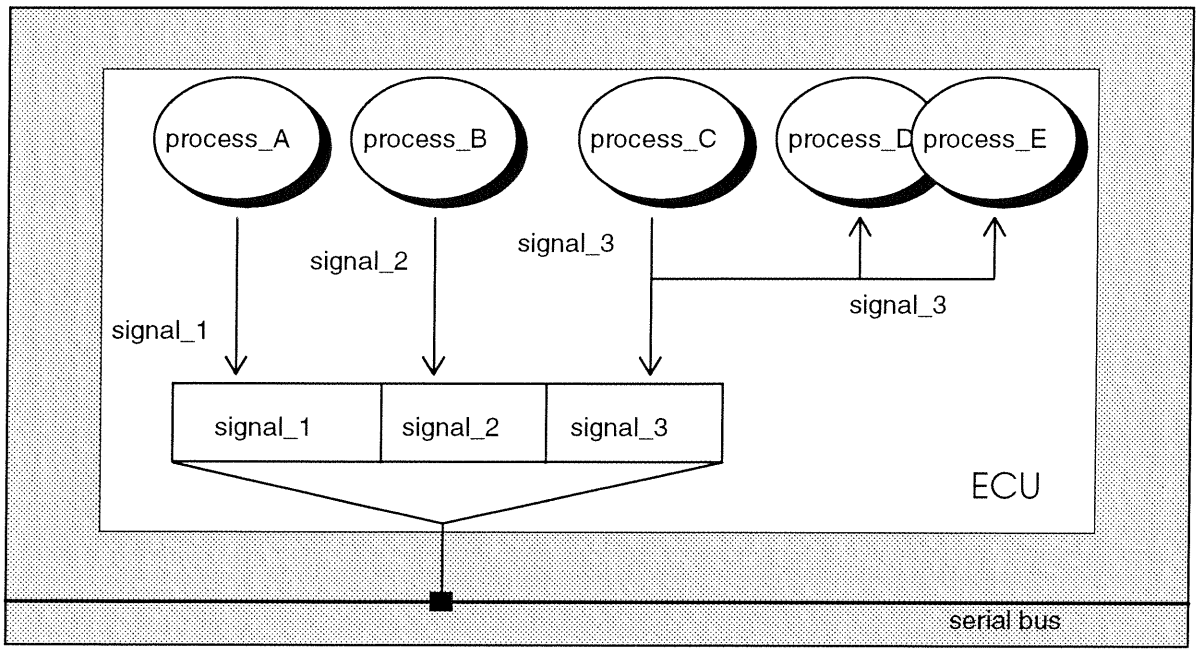

Figure 2: Communication in an ECU

Instead of transmitting each signal on its own these information units may be put together into one bus packet. The receiving ECU then of course has to know at what position in the packet the desired signal is located. This bus packet may be received by different ECUs with different interests in the contents. So one ECU may require signal_1 and another one may require signal_2.
This kind of „piggybacking“ of course will only work when timing requirements of the transported signals are respected. This means that you can only group signals together that have similar or multiple transmission rates; a signal that is expected every 20ms may be grouped together with a signal that is sent every 10ms. In addition some „trigger" information must be added to make the controller send a packet only if the specified trigger conditions have come true.

4 ODL: An Open Description Language to describe communication between and within ECUs

As can be seen in the previous sections the description of a communication scenario may be seen from different views. One goal of our work was to introduce an easy to use language that supports an abstract specification of the scenario. Implementation details should be hidden from the programmer ("user"). For this purpose we introduced ODL, the Open Description Language. It uses only a very small set of intuitive syntactical elements and allows users to specify their communication scenarios in an abstract way.

Some examples of the language elements are:

- signals — that are the logic information units used in any kind of communication
- processes — that produce and consume signals (if only regarding communication purposes)
- containers — which are elements to transport signals from or to processes
- connections — elements that connect processes with container objects
- drivers — that allow an easy and user-defined attachment of different hardware to the system

The basic idea of the language is that signals move between processes. A process produces a signal and sends it via a connection to a container. The container then is responsible for the transportation of the signal (by using some kind of underlying hardware). At its destination the signal reaches the other process again via a connection.
Each of the language elements/objects may have additional attributes that specify the particular instance of this object, e.g. the type of communication (IOC, NWC or IPC) being specified in a container object. Further objects are used to allow timing and exception handling features.
ODL also features the possibility to specify user-defined data structures and hardware descriptions. For a full language description and reference see [3].
A short example of an ODL-description can be found in the appendix.

5 Producing code for different target systems

As already mentioned, hardware can be very heterogeneous in automotive networks. Different types of CPUs and microcontrollers are used in ECUs that have to be supported. For this purpose a compiler translates the description language into data structures for different target systems (see figure 3, the highlighted area).
Due to the object-oriented and therefore modular architecture of the implemented compiler it is easy to add further code generators. Among the implemented generators are those for Intel, Motorola and Philips CAN-chips. But there is no problem in porting the output also to other bus systems.
Generated code uses the services provided by a special communication software and an underlying real-time operating system that are already implmenetd and were both parts of our first efforts.

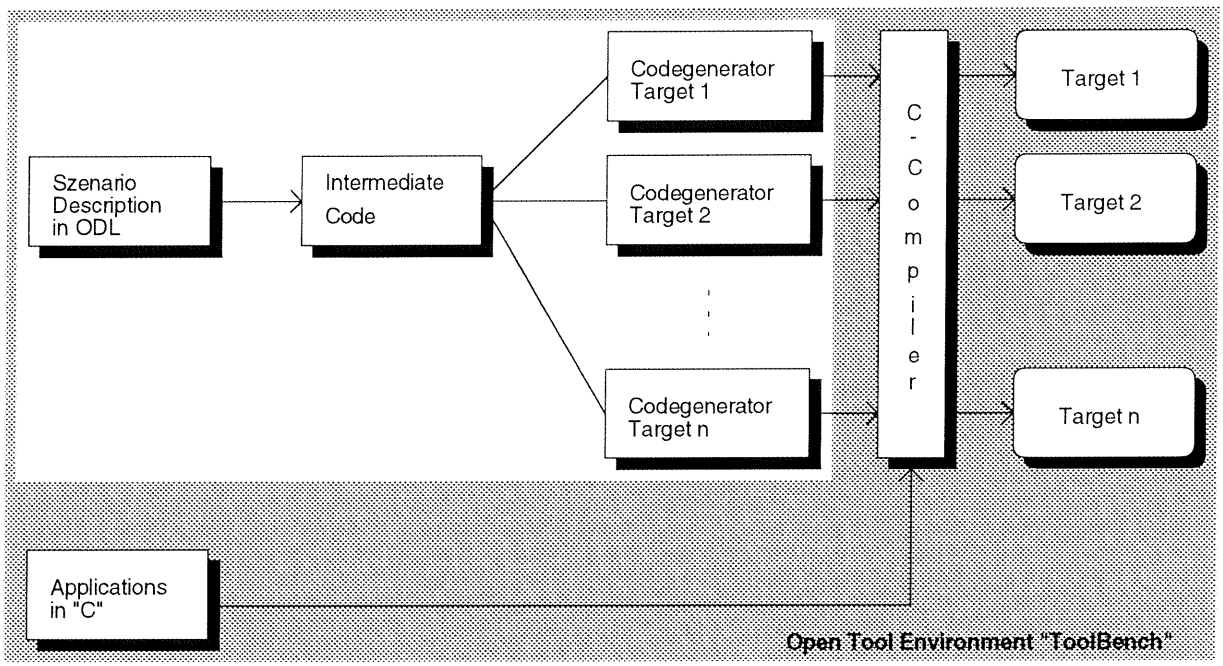

Figure 3: Code generation for different targets

The compiler itself is one part of the design framework "Open Tool Environment - Tool-Bench"[4] that integrates generation of communication- and application-code with testing facilities. ToolBench also allows the integration of different other tools like semantic-checkers and graphical output facilities.
A simulation of the system including a complete CAN-bus model for all kinds of controllers is subject of on-going research at our institute at the university of Stuttgart. This simulation will help to guarantee performance requirements and show the deterministic behaviour for a specified net.

6 Conclusion

In our paper we presented an easy to understand and easy to use description language (ODL) to specify communication scenarios in vehicle networks. We derived this language from communication requirements and general (communication) needs in automotive systems. It is therefore used to specify the scenarios and to serve as input for a sourcecode compiler that produces code for different target systems. This approach provides a standard description solution for nowadays heterogeneous automotive nets. Providing simulation equipment is part of our on-goin research.

7 References

[1] Stümpfle, M. "*Communication Mechanisms in Automotive Systems*", Proceedings of the 3rd Open Workshop on HS-Networks, Paris 1993

[2] Stümpfle, M. "*Konzeption und Erstellung von Steuergerätesoftware*", internal paper, IND 1993

[3] Stümpfle, M, Eberspächer, M. "*Open Description Language*", IND 1993

[4] Stümpfle, M. "*Open Tool Environment*", internal paper, IND 1993

8 Appendix

The following example of an ODL-Description reflects the scenario introduced in section 3.2. Only communication aspects are taken into consideration. Further description possibilities like process priorities, process memory allocation or completion routines for synchronisation purposes are not displayed. As you can see include-technics are used to insert the hardware (driver) description.
Note: keywords for new instances are highlighted for easier reading. The IO-signal to receive the engine speed is not displayed in this example.

```
/*********************************/
/* odl description example       */
/* M. Stuempfle                  */
/* IND, University of Stuttgart */
/*********************************/

#include "can82527.odl"

ecu example {
       process = process_A;
       process = process_B;
       process = process_C;
       process = process_D;
       process = process_E;
}

/* signals in ecu ***************/

signal signal_1;
signal signal_2;
signal signal_3;

/* processes in ecu *************/

process process_A {
       signal = signal_1;
}

process process_B {
       signal = signal_2;
}

process process_C {
       signal = signal_3;
}

process process_D {
       signal = signal_3;
}

process process_E {
       signal = signal_3;
}

/* net communication ************/

container NetWork {
       signal = signal_1;
       signal = signal_2;
       signal = signal_3;
       driver = can82527;
       type = nwc;
}

connection con1 {
       signal = signal_1;
       direction = send;
       task = process_1;
       container = NetWork;
}

connection con2 {
       signal = signal_2;
       direction = send;
       task = process_1;
       container = NetWork;
}

connection con3 {
       signal = signal_3;
       direction = send;
       task = process_1;
       container = NetWork;
       trigger = true;
}


/* interprocess communication ***/

connection con4 {
       signal = signal_3;
       direction = send;
       task = process_C;
       container = Internal;
}

container Internal {
       signal = signal_3;
       type = ipc;
}

connection con5 {
       signal = signal_3;
       direction = receive;
       task = process_D;
       container = Internal;
}

connection con6 {
       signal = signal_3;
       direction = receive;
       task = process_E;
       container = Internal;
}
```