

Software Development for Multiple OEMs Using Tool Configured Middleware for CAN Communication

Pascal Jost

IAS, University of Stuttgart, Germany

Stephan Hoffmann

Vector CANtech Inc., USA

Copyright © 2000 Society of Automotive Engineers, Inc.

ABSTRACT

Years ago car manufacturers started using reusable software components for the development of ECU applications. As communication between ECUs is not a competitive advantage, there has been a lot of standardization efforts, especially in Europe. Whereas the standards for Operating Systems and Network Managements are widely accepted and used, all major OEMs created their own versions of Communication Modules, which presents a major roadblock for OEM independent ECU development. We show how the OEM specifics can be hidden by using a tool configured middleware.

INTRODUCTION

Today's vehicles are growing to become more and more complex systems. The functionality of a modern vehicle is not dominated by mechanical components anymore. Electronic Control Units (ECU), sensors and actuators became irreplaceable parts of a vehicle. They are responsible for controlling the power train, the chassis and the body of a vehicle. The ECUs are spread across the whole vehicle. For example the dashboard, the air condition, the fuel injection, the automatic transmission, and even minor features like the power windows are controlled by ECUs. An ECU itself usually consists of a microprocessor, memory (ROM and RAM), some additional chips to connect the peripherals, and software.

To control and synchronize the distributed software within a vehicle, communication between the ECUs is indispensable. Data as well as commands and state information needs to be exchanged. Therefore, a network is essential. Usually a modern vehicle contains more than one network. This is because different parts of a vehicle have different requirements for the communication. The power train for example needs a fast exchange of few

data. On the other hand, the body needs the exchange of a lot of data, but at a lower rate as in the power train. These different requirements for the data exchange can't be met by one network. Therefore, several network segments are used for several tasks. Most of the OEMs are using Controller Area Network (CAN) [1] which can be adapted to the different requirements.

To ensure that the ECUs within a vehicle understand each other, they need to use the same language. The language for the communication by a network is defined by protocols organized in layers. Every ECU has to handle the exchanged data according to these protocols. Therefore, in every ECU a piece of hardware and a piece of software is needed to do this job. This software is provided by the OEMs and every ECU vendor have to use it. This works fine, but every OEM developed its own driver and protocol stack. This leads to ECUs which are incompatible to ECUs of other OEMs.

The fast development cycles of vehicles and the rising costs of development result in the need of reuse of application software or entire ECUs. It's therefore necessary to adapt the application to the requirements of several OEMs. We will see why this is hard to achieve, even if the requirements are quite similar.

This paper discusses the problems of OEM independent application software development and shows some possible solutions.

PROBLEMS OF SOFTWARE DEVELOPMENT FOR MULTIPLE OEMS

AUTOMOTIVE NETWORK

A network consists of a physical media to transport information and several nodes which receive and transmit data via the network. To understand each other

the nodes have to talk the same language and they need a mechanism to synchronize their communication. Therefore, the nodes use protocols arranged in layers. We will use a layer model with just four layers as shown in figure 1 (CAN-Hardware, CAN-Driver, Interaction Layer, Application). First, we collect all the hardware specifics to one layer. The hardware will provide the data to the upper layer by the use of bus frames. Second, we encapsulate this hardware layer with a driver. This driver should provide the exchanged data as messages. Third, we use a layer as the interface to the application which processes the data for the easy use of the application. It provides data access in a signal-oriented manner. Fourth, we need an application layer to handle the data.

Let's consider the whole vehicle again. Until now we have a set of ECUs connected by a network. These ECUs exchange data by the use of protocol layers as described above. For each signal a transmitter and one or more receiver exists. To specify and document the communication within a whole vehicle we need to describe each signal with its transmitter and its receivers. Further we need to describe the data type and the transmission mode of the signal. This information can be described using a matrix. We will refer to this matrix as the network database. By sorting this matrix by items of a single ECU we derive the interface between this ECU and the network.

APPLICATION CONTEXT

If we consider a PC with an Internet connection, all the tasks of the several network layers are done by standardized software components. Our goal is to use standardized software components for the communication of the ECUs, too. We expect shorter development cycles, less costs, reduction of the workload of the application and higher quality. The application developer can concentrate on the development of the application and does not need to invent the communication components over and over again. However, the use of standardized communication components also causes some problems. The major problem is that each OEM defines its own standard and the application developer must adapt the application to each OEM to reuse it.

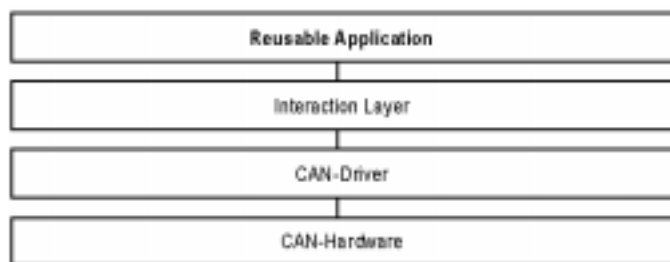


Figure 1: CAN Communication Modules in the Application Context

Because most of the OEMs use the Controller Area Network (CAN) [2][3][4] as a inexpensive and reliable network to connect the ECUs of their vehicles, we will

use CAN as the example throughout this paper. A simplified picture of the communication layers of an ECU is shown in figure 1.

The CAN driver provides a mostly hardware independent interface to the higher communication layers, thus enabling the hardware independent implementation of these modules.

The Interaction Layer [5] is responsible for data exchange by the means of process data. This data is provided to or by the application in a signal-oriented manner. It is independent of the underlying bus system. Because we will focus on the Interaction Layer later, more details about this layer are given in the next sections.

The Reusable Application represents the functionality of the ECU. It is responsible for the content of the exchanged data.

FEATURES OF THE INTERACTION LAYER

The Interaction Layer hides all the communication details from the application to ease its workload. In case of transmission the application just needs to pass the data to be transmitted to the Interaction Layer. After this the Interaction Layer does the rest to get the data on the network. In case of reception the Interaction Layer will store the received data and notify the application about the arrival. Then the application can decide what to do with the updated data.

However, the data is represented in a signal-oriented manner. A signal is an entity of data from the application point of view. I.e. the Interaction Layer meets the communication requirements of the application, not the other way round. E.g. a signal called *Tx_VehicleVelocity* is contained in the message *CAN_VMD* which is transmitted by the bus frame with the *ID 253*. In this example it is easy to deduce the content from the name of the signal. The use of signals also enables the exchange of data which is independent of the length of a bus frame or a message. Short signals could be collected and combined to one message. This will reduce the network traffic.

The Interaction Layer supports several transmission modes. The transmission modes are responsible for transmitting the signals on predefined events. For example, events could be generated by an elapsed timer, updated data of a current signal value compared to a constant or the old signal value. Further, the Interaction Layer handles timers and counters, used e. g. for timeout monitoring of periodic signals. In either case the application does not need to know how the data is transmitted or received by the lower communication layers. The data structures of the application are independent of the data structures of the CAN driver or

the bus (i.e. messages, bus frames). The result is a higher reusability of the application software.

ADAPTATION OF THE INTERACTION LAYER

As depicted in the chapter above the Interaction Layer provides access to data for transmission or reception. Therefore an Application Program Interface (API) is used which depends on both the ECU and the vehicle. Of course, every ECU in a vehicle needs to exchange different signals. I.e. every ECU in a vehicle needs a different unique API. Further, each OEM has its own naming convention and signal names.

To adapt the API of the Interaction Layer to an ECU for an OEM we need a description of the communication requirements of a vehicle. This is stored in the network database which is provided by the OEM. So we can derive the function names for the API directly from this network database. This work is done by a tool. All the function names of an API will be generated by this tool using a network database. Figure 2 shows the generation process including the sources.

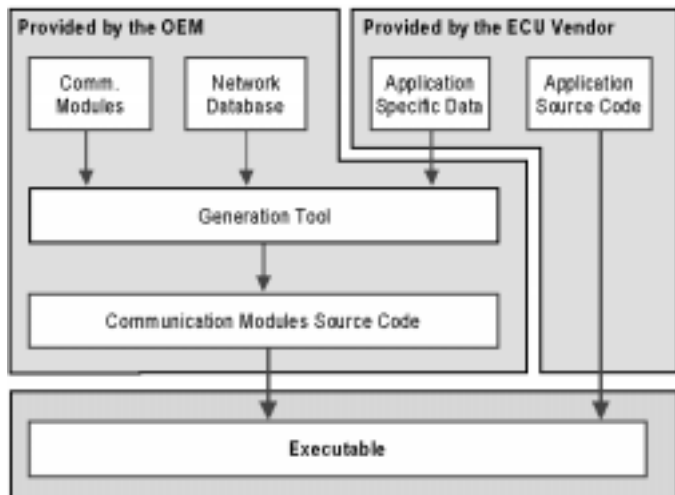


Figure 2: Generation Process for Automotive Applications.

The adaptation of the Interaction Layer to each ECU of each OEM results in a set of similar, but different APIs. The developer of applications for these ECUs won't like this, because they need to adapt their applications to each OEM. This can't be done by a tool and means that the applications become unique and can't be reused in any way. In the following chapter we will present a suggestion how this problem could be solved.

PERPARING FOR REUSABLE DEVELOPMENT OF AUTOMOTIVE APPLICATIONS

MULTIPLE VEHICLE MANUFACTURERS

Vendors of ECUs or applications for vehicles are interested in selling their product to more than one OEM. It's obvious that for example the ECU of a rear door in car x has to provide similar features as the rear door of any other car y. This is true not only for several car lines of one OEM but also for vehicles of several OEMs. So, why is reuse so difficult?

One reason for this is the fact that different OEMs use different communication modules. The communication method will be different in every car line. The OEMs provide network databases which describe the communication of the ECUs, as mentioned above. These network databases include message IDs, the names of the transmitting and receiving ECUs, signal names, transmission modes, transmit periods and so on. But this is not the only thing. Each car manufacturer also provides its own communication modules which have to be used by the vendors of ECUs. Possibly even different types of bus-systems could be used.

This results in two unsolved main problems in developing reusable applications. First, every car manufacturer provides its own communication modules with its own API and features. Second, every car line has its own network database which describes the communication behavior of this unique model.

In this paper we will concentrate on the Interaction Layer (IL) as the communication interface. On the one hand, the ILs of several OEMs have a lot in common, so OEM-independent development seems achievable. On the other hand the remaining differences affect all parts of the application and a commonized API would be of the greatest benefit.

OVERCOMING THE OEM DEPENDENT SOFTWARE DEVELOPMENT

WRAPPER CONCEPT

The idea to solve the problems described above is to insert a wrapper layer between the application and the Interaction Layer. This wrapper layer should encapsulate all car manufacturer specifics of the Interaction Layer.

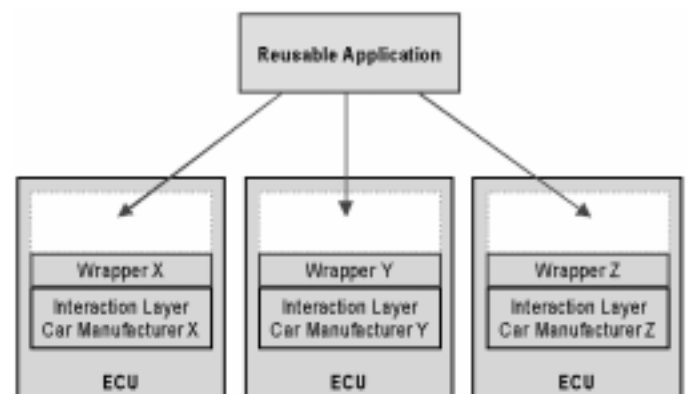
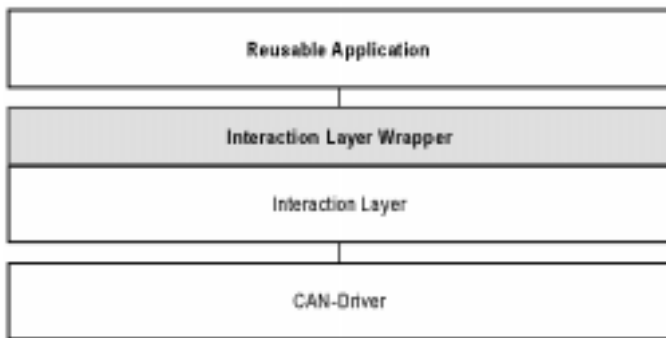


Figure 3: Wrapper Concept

Each car manufacturer provides an Interaction Layer and a network database. The network database will be different for each car line. As shown in figure 3 the vendor who wants to implement a reusable application has to define its own independent interface. To interchange this application together with its interface the vendor needs to translate this interface to the interfaces provided by the Interaction Layers of the OEMs.

ARCHITECTURE

Figure 4 shows an architecture which allows for the development of reusable applications, but it won't guarantee it, as the application itself has to be modified in order to be reusable. The architecture consists of an application core and some wrapper modules which encapsulate the OEM specific and hardware specific modules. The main feature of this architecture is to hide all manufacturer specific interfaces by using wrapper



modules.

Figure 4: CAN Communication Modules Including a Wrapper Layer

ADAPTATION OF THE INTERACTION LAYER

The main task of the communication wrapper module is to encapsulate the Interaction Layer of different OEMs and different car lines. As described in the previous sections the application developer will get a network database, an Interaction Layer and a generation tool. With the help of a generation tool the developer can adapt the Interaction Layer to the network database and the application. But this adaptation won't be enough to build an application which is independent of the OEM and the type of vehicle. It just builds an API for the application which depends on the names and data types defined in the network database.

The developer who designs a reusable application needs to use her/his own signal names, pre- and postfixes and further configurations. The wrapper module as described above is needed as an interface between the API which is generated by the generation tool and an API defined by the developer.

As an interface between the application and the Interaction Layer the wrapper module has to meet the following requirements:

- Translate function names used by the application to function names used by the OEM.
- Convert data types, scale values and adapt units
- Adapt parameters of callback functions for the notification of the application
- Adapt the API to control the communication modules (e.g. *TxStart*)
- If the communication modules of a car manufacturer does not support a feature which is needed by the application, the wrapper module has to provide an extension to these communication modules to meet this feature.
- If the application does not meet a requirement of the OEM the wrapper module should provide an extension to the application to meet this requirement. This could cause some problems, if an application can't supply necessary data.

WRAPPER DEVELOPMENT

Overview

For the reuse of an application for different OEM it is not enough to have a wrapper module. The developer has to keep in mind that the application has to meet the requirements of all OEMs.

Software development for multiple OEMs has to deal with two problems. First, the application needs to be developed for reuse. I.e. OEM-independent applications have to meet the requirements of all of the considered OEMs. This first problem is out of the scope of this paper. Second, the interface between the reusable application and the Interaction Layer needs to be adapted to every considered vehicle. A reusable application which should be independent of any OEM has to be adapted to the Interaction Layers of several vehicles. The API of the Interaction Layer, of course, depends on the OEM and the type of vehicle. Therefore, this paper proposes a method to solve these two problems in a structured way. The method will support the developer to create Interaction Layer wrappers. An Interaction Layer wrapper works as a kind of translation module between the reusable application and the Interaction Layer as described in the sections above. Therefore we will need to develop a wrapper for each of the considered car line.

We assume that the developer of the application has a collection of requirements, specifications and network databases provided by several OEMs. This is the starting point of our method and the goal is to convert this into a set of structured requirements for our Interaction Layer wrappers.

We provide a method to reach this goal in six steps. First, we suggest to list all the signals used for data exchange and sort them by features. Therefore it is necessary to look at all the considered vehicles. Second, we suggest

to collect as much information about the signals as possible. It's because we need a detailed description of the considered interfaces. Third, it is necessary to find the commonalities and the variabilities of the several interface descriptions. Within this step we want to find a set of signals of all of the considered interfaces. Fourth, we need to find a common interface for the reusable application. This common interface needs to provide all the data which is necessary to satisfy all the requirements of the considered interfaces. Fifth, we are now prepared to derive the requirements for the Interaction Layer wrapper. As a translator each wrapper has two interfaces. One to the application and one to the OEM's Interaction Layer. For both we have a detailed description. So, we could derive the requirements for the translation from the description of the two interfaces. Last but not least, we need to design wrappers which meet the derived requirements. These six steps are summarized below and will be described in the following sections.

1. Collecting signals of the considered vehicles and sorting them by features
2. Collecting detailed information about the signals
3. Exploring the commonalities and the variabilities of the considered interfaces
4. Deriving the interface description for the reusable application
5. Deriving the requirements for the Interaction Layer wrappers
6. Designing the Interaction Layer wrappers

Collecting Signals of the Considered Vehicles and Sort them by Features

The scope of an Interaction Layer wrapper is the adaptation of the application API and several Interaction Layer APIs. In this step we want to look at several Interaction Layer APIs. Therefore, we start with the analysis of the network databases which contain the description of the data to be exchanged between ECUs. Of course, we won't need the whole information of the network databases, but only the incoming and outgoing signals of our ECU, which we will collect for each of the considered vehicles. To document the results of this step we use a table. An example can be seen in figure 5.

Feature	Car 1:		Car 2:	
	message	signal	message	signal
Window Control	wis_bl	bl_move	-	-
Window Control	wis_bl	bl_up_man	wc_rear_left	ri_up
Window Control	wis_bl	bl_down_man	wc_rear_left	ri_down
Window Control	wis_bl	bl_close_auto	wc_rear_left	ri_close
Window Control	wis_bl	bl_open_auto	wc_rear_left	ri_open
--	--	--	--	--
Window State	wis_bl	bl_opened	-	-
Window State	wis_bl	bl_closed	-	-
Window State	wis_bl	bl_locked	ws_rear_left	ri_state
Window State	wis_bl	bl_pos	ws_rear_left	ri_position
--	--	--	--	--

Figure 5: Collected Signals Sorted by Features

Each set of collected signals represents an interface description of a considered ECU. The entire group of sets represents the interface description of our reusable application. However, this interface description of the reusable application is unstructured and seems to be useless in this form. We need a way to structure and sort the sets of collected signals. Therefore some criteria will be useful. We suppose the features of the ECU would be a good choice as criteria. The criteria could also be used to find common terms and will be useful to compare the interfaces. For example the ECU in the rear left door provides the feature to control the power window of this door. The features could be retrieved from the data which is transmitted by the collected signals. However, it is possible to select features of different abstractions. For example, if a signal *rl_up* is used to control the movement of the window of the rear left door, it is useful to relate it to the feature *Window Control*. But it could also be related to the feature *Lift Power Window* which is a sub-feature of *Window Control*.

The list of features represents the interface between the wrapper module and the OEM's Interaction Layer. At the same time this list will give us requirements for the OEM independent interface of the application which we want to develop. We believe that the interface becomes independent by considering all the OEMs and all the car lines.

Collecting Detailed Information About the Signals

Until now, the only information we have about our interfaces results from the network databases. To develop a wrapper we will need further, more detailed information. For example, a network database contains a signal called *rl_winpos* of the length of 8 bits. Maybe it also includes a remark, but usually we can't get further information about the unit, scale, offset and so on of the physical value from the network databases. Therefore we need to consult specification and requirement documents to derive this information. This information, for example, the interpretation of a signal as a physical value, can be collected in a set of detailed interface descriptions of the Interaction Layers of the considered vehicles. We list the attributes which turned out to be most valuable for us in figure 6.

Attribute	Example
message name	win_of
signal name	bl_aba
signal size and type	8 bit, can int8
unit of physical value	%, cm, inch
range of physical value or enumeration	{0,200}
scale of physical value	1 = 0,5%, 1 = 0,25cm
timeout	yes
default value	0
timeout default value	0
interpretation of physical value or boolean value or a list of interpretations for an enumeration each as plain text	0% = closed, 100% = opened

Figure 6: Attributes for the Detailed Description of an Interaction Layer

Exploring Commonalities and Variabilities

We now have all the information we need to compare the interfaces of the Interaction Layers of the considered vehicles: a set of features derived from the network databases and a set of detailed interface descriptions of the Interaction Layers. The result of the comparison should be a list of common and different features and signals.

To get the common and different features we suggest to connect the set of interface descriptions. Because they are sorted the same way it should be easy to compare them feature by feature. Within this process we will find features which contain signals of different names. However, the signals could have the same task, but they fulfill this task in various ways. Other signals possibly are provided only by some of the considered interfaces or by all of them the same way. This results in alternative, optional and mandatory features and signals, see [7] and [8]. The goal of this step is to build a list which shows an intersection set of all features and signals as well as several subsets. For an example see figure 7. The union of all the features of the considered interfaces will represent the features of the application interface.

Feature	Car 1: (signal)	Car n: (signal)	Type of Feature
Window Control, motor on	bl_motor	-	optional
Window Control, win. manual up	bl_up_man	r1_up	mandatory
Window Control, win. manual down	bl_down_man	r1_down	mandatory
-	-	-	-

Figure 7: Classified Set of Features

Deriving the Interface Description for the Reusable Application

As a result of the comparison of the interface descriptions we can build the interface description of the application. The API should cover the requirements of all the interface descriptions we derived. The interface description of the application should contain information about the type and the interpretation of the signal. Based on this information the application developer could build an application which has a common interface to all of the considered vehicles and is therefore independent of the communication.

Feature	Name of Variable	Size	Unit	Range	Scale	Interpretation
Window Control	up_man	1 bit	-	0,1	-	1 = move win up
Window Control	down_man	1 bit	-	0,1	-	1 = move win down
Window Control	close_auto	1 bit	-	0,1	-	1 = close win
Window Control	close_auto	1 bit	-	0,1	-	1 = open win
...
Window State	opened	1 bit	-	0,1	-	1 = win is opened
Window State	closed	1 bit	-	0,1	-	1 = win is closed
Window State	locked	1 bit	-	0,1	-	1 = win is locked
Window State	position	8 bit	%	{0-200}	1 = 0,5%	0 = closed
...

Figure 8: Interface Description of the Application

This step comprises the identification of the interface of the application to the rest of the vehicle. Therefore, we investigate the union of all of the features. Even if a feature will be represented by different signals or physical values by the different OEMs, it could result in a single signal for the application. For example, if one OEM needs a signal representing the speed of the vehicle in mph, and another OEM needs the same in km/h, a common feature – velocity in this example – should be determined. Therefore, the interface of the application needs to provide the feature velocity of the vehicle in an appropriate form. A set of signal values from which we can derive the velocity of the vehicle would be alright, too. The result of this step should be a table like the one shown in figure 8. The table describes the interface of the application which is independent of any considered type of vehicle. It should contain the same attributes like the table of features in the first step.

Deriving the Requirements for the Interaction Layer Wrapper

The task of this method is to guide the wrapper developer, not the application developer. So we will concentrate on the Interaction Layer wrappers again. The goal is to develop an Interaction Layer wrapper for each of the considered vehicles. Until now, we have got an interface description of each of the considered Interaction Layers and a single interface description of the application. In terms of a layer model we have the description of the upper interface to the application and the lower interface to the OEM's Interaction Layer.

As depicted, the wrapper modules could be seen as a kind of translator between the reusable application and an Interaction Layer specific to an OEM. So, as every translator a wrapper has got a source language and a destination language by the means of a source interface and a destination interface. We need to know which source signals should be translated to which destination signals. This information will result in the requirements for the several Interaction Layer wrappers.

An appropriate way to describe the requirements could be a table which contains a description of the source and the destination interface. The goal of the table is to list the interfaces face to face to get a good overview of the requirements.

The translation has to follow some rules. As known from language translation we need rules to translate the words (dictionary, database) and the grammar (functionality of IL). The rules may differ depending on the direction and the language in which you translate. The same problem appears during the development of the Interaction Layer wrappers. We need rules to translate from the application to an Interaction Layer and vice versa.

Designing the Interaction Layer Wrappers

All the outgoing signals of the Interaction Layer wrapper have to be derived from incoming signals. Therefore a set of rules and algorithms for the translation is needed to convert the signals. The scope of this step is to find such a set of rules and algorithms. The result of this step are tables describing the differences between the application and each of the vehicles considered

Kinds of Differences	Suggestions for Solutions
Different function names	This is a very simple kind of difference. For example macros could be used to solve this kind of problems. If this does not work, we use a simple function.
Different physical values	The reason for this problem is the different interpretation of the value of a signal. Therefore a type conversion of a physical value could be used. Sometimes the signals even have a different number of bits. This actually could result in a loss of data, but could be solved by a scaling factor.
A signal is not provided by the OEM	If a signal was not included in a network data base of a considered vehicle, there will be two possible solutions for this problem. First, the developer could try to calculate the signal by using combinations of other signals. Second, the developer could ask the OEM for providing this signal.
A signal is not provided by the application	If a signal was not provided by the application, there will be two possible solutions. First the developer could try to derive the signal's value by other signals. Second the developer of the application could add this feature to the application
A feature is not provided by the used Interaction Layer	This means the application wants to use this feature, because a Interaction Layer of an other OEM provides it. So the wrapper has to fulfill this task of the Interaction Layer.
A feature is needed by the Interaction Layer but is not provided by the application	This means that the developer of the application decided not to support a feature which is needed by some Interaction Layers of the considered vehicles. So the wrapper has to fulfill this task of the application. This problem has to be solved individual by functions of the wrapper.

Figure 9: Overview of Translation Rules

The requirements for the wrappers can be categorized. I.e. the requirements resulting from the same kind of differences could be solved by similar rules or algorithms. Therefore, we list some common kinds of differences in figure 9 and suggest a solution for each of them.

To illustrate how this can work we choose the rule "Different Physical Values" as a simple example. If the transmitter and the receiver of a physical value do not use the same offset, scale or unit, a converter is needed. This converter algorithm will be the Interaction Layer wrapper.

A signal value which represents a physical value implicitly contains an offset, scale, a unit and a range as described below.

$$\begin{aligned}
 &physical_value = offset + signal_value \cdot scale \cdot unit \\
 &signal_value \in range \\
 &range = [range_lower_border, range_upper_border]
 \end{aligned}$$

To convert signal values the scales and the units of the incoming and the outgoing values have to be considered. The following formula will show how a linear conversion with an offset could look. The range is not considered in this example.

$$x_{out} = x_{in} \cdot \frac{unit_{in} \cdot scale_{in}}{unit_{out} \cdot scale_{out}} + \frac{offset_{in} - offset_{out}}{unit_{out} \cdot scale_{out}}$$

By this formula every conversion of the same kind could be done. To make it a bit easier to design a wrapper it is useful to categorize the problems and find common solutions like the one described in the example above.

TOOL SUPPORT

Overview

The Interaction Layers are configured by a tool and a lot of API functions, such as read and write functions for signals are automatically generated. This is true for most OEMs, so why can't the wrapper layer be configured and generated by a tool? The answer is simply that to configure the Interaction Layer you need to read only one database, whereas to configure the wrapper layer you have to consider at least three: the ones provided by the OEMs and the one containing the names used for the application.

Furthermore, you need a means to map signals in one database automatically to signals with different names in another database. This mapping can't be automated.

This means that a tool can not be used to establish a translation dictionary. But it can still be used to help with the 'grammatical intricacies' during the 'translation process' from one IL to another. The Interaction Layers are OEM-specific, but application-independent, which means that the dictionaries change from module to module and from car line to car line, but the rules remain the same.

Requirements for a Tool

We will go through the method established above step by step to derive requirements for a tool. For each step we will discuss what can be automated and what not. The steps are numbered the same way as in the previous section. These numbers are also used in figure 10 which will provide an overview of the method.

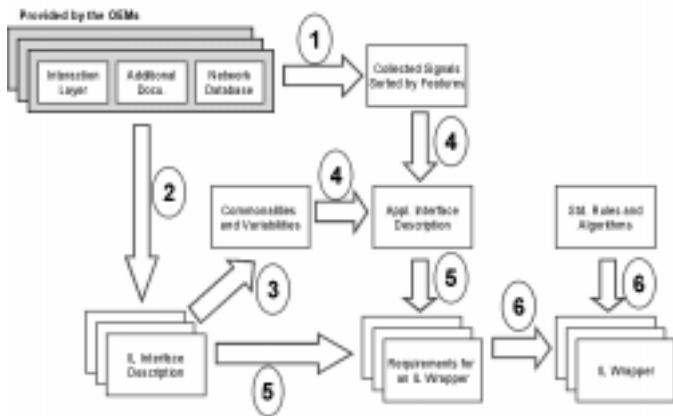


Figure 10: The Method of the Wrapper Development

1. Collecting signals of the considered vehicles and sort them by features

The tool needs to manage the import and the connection of the network databases of the considered vehicles. After this the tool should sort the signals by names and ask the developer for the features to which a signal belongs to. On demand the tool should sort the signals by the assigned features

2. Collecting detailed information about the signals
The tool asks the user for all the information about signals which can not be found in the database.

3. Exploring the commonalities and the variabilities of the considered interfaces
The tool could try to detect the commonalities of the considered ILs. Of course, the tool can't resolve variabilities. Therefore, it should ask the developer for help to assign signals of different names and common features. This has to be built into the tool for all OEMs under consideration.

4. Deriving the interface description for the reusable application
The collected data could be used to provide a complete list of features to the developer and to suggest a data type by the tool. The tool just needs to prompt the developer for appropriate names for the signals of the interface to the application. Alternatively, the user is not prompted for names but asked to make connections between the OEM database(s) and the application specific database.

5. Deriving the requirements for the Interaction Layer wrappers
From the collected data the tool should derive the interface descriptions of the application interface and the IL interface.

6. Designing the Interaction Layer wrappers
By the type of the variabilities the tool should suggest translation rules and algorithms to the developer. If variabilities can not be resolved automatically by the tool, the user is prompted.

Now the tool is able to completely configure and generate the wrapper layer.

CONCLUSION

We propose the use of a wrapper layer to enable OEM-independent development of an application. The wrapper layer encapsulates the API of the Interaction Layer. This results in an OEM-independent interface to the application, i.e. the application developer has the possibility to create a reusable application. To achieve this the wrapper layer needs to be adapted to the Interaction Layer of all car lines considered.

To enable a structured wrapper development we introduced an easy to use method. It supports a developer transitioning from the network databases of several vehicles to a common interface of the application and a wrapper layer. The developer can proceed step by step to a design which considers all the car lines.

The problem of adaptation seems to be shifted to the wrapper layer, but not solved. It's right, the problem was just shifted, but we now have the chance to solve it much more easily. The result of the wrapper idea is a reusable, OEM independent application and an OEM dependent wrapper layer.

We are sure, that a tool to assist the developer for the creation of a wrapper layer is feasible and we have the fundamentals to do this. Such a tool can ease the workload of the developers and shorten the development cycle time.

Of course, the usage of a wrapper layer increases the need of RAM and ROM as well as the CPU load. How much the usage of a wrapper layer will cost depends on the considered Interaction Layers. This is the price of reuse we have to pay for short development cycles.

REFERENCES

1. "CAN Specification Version 2.0", Robert Bosch GmbH, Schwieberdingen, Germany
2. K. Etschberger: "CAN; Grundlagen, Protokolle, Bausteine, Anwendungnen", Konrad Etschberger, Hanser, 1994
3. W. Lawrenz: "CAN System Engineering – From Theory to Practical Application", Springer Verlag New York, Berlin, Heidelberg, 1997
4. B. Baudermann: "CAN-Kommunikation als Softwarekomponenten für Kfz-Steuergeräte", Embedded Intelligence 2000, Design&Elektronik, 2000
5. "Interaction Layer, User's Manual Version 1.4", Vector Informatik GmbH, Stuttgart, Germany
6. "OSEK/VDX Communication, Version 2.2", 1999
7. C. Chan et al.: "Feature Oriented Domain Analysis (FODA). A Feasibility Study". SEI, Carnegie Mellon University, 1991
8. U. Eisenecker, K. Czarnecki: Generative Programming, Addison Wesley, 2000

CONTACT

Stephan H. Hoffmann
Vector CANtech Inc.,
stephan.hoffmann@vector-cantech.com