TITUS – A Graphical Design Methodology for Embedded Automotive Software

Ulrich Freund, Alexander Burst, ETAS GmbH Stuttgart

Abstract

Vehicle body electronic software has reached a level of complexity and cost that methods focusing on re-use and seamless vehicle integration of networked ECUs are being heavily embraced. In this paper, the TITUS methodology is introduced showing that vehicle functions can be re-used over several physical ECU networks, i.e. several vehicle models.

1 Introduction

Nowadays it becomes obvious that the functionality of a vehicle’s electronics should be independent of the ECU architecture. The recombination of functions once developed with new demands is one of the crucial items for a quick reaction in the dynamic passenger vehicle market. Furthermore, re-use is also indispensable to keep software development costs within acceptable limits. Re-use has to be applied to design as well as validation effort to obtain maximum productivity gains. Figure 1 shows several functions that can be mapped to different vehicles.

The complexity of automotive software is passing the critical point where traditional approaches tend to fail. Development partnerships between vehicle manufactures and suppliers require a methodology defining lean interfaces between algorithms which provide easy interchangeability of software by keeping the context of its environment transparent.

Figure 1: Re-use of Software in Different Vehicles

2 The TITUS Methodology

2.1 UML and Embedded Automotive Software

Automotive body electronics are characterized by a distributed architecture, highly interconnected functions and late changes of functionality prior Start of Production (SOP).

---

1 ECU is an abbreviation for Electronic Control Unit, i.e. the embedded system controller in a vehicle.
Running software on ECUs imposes constraints on the software design itself. Typical design constraints are resource limitations and hard real-time requirements resulting from safety regulations. However, the number of service requesting entities do not change dynamically, i.e. the number of doors usually does not change during a vehicle’s lifetime. Furthermore, CPU peripherals are allocated to dedicated signals.

2.2 The TITUS Design Methodology

Research work on the TITUS design methodology started at DaimlerChrysler in 1994 and was first published in 1997 [1]. Its goal is to separate the design of vehicle software functionality independent of the hardware. This leads to reusable components that can run on several ECUs. It resembles in many cases to the ROOM methodology but differs considerably in details mainly to make an ‘actor-oriented’ approach suitable for ECU-Software. A detailed comparison between the TITUS- and the ROOM methodology is given in [7].

Being an architecture description language [5][8], the TITUS design methodology comprises design primitives for structure and behavior – surplus modeling means reflection its application domain: embedded automotive software.

The process class is the core modeling primitive. Though resembling to some extend to a class in UML it differs by

- being an active object, which means it has behavior that might execute concurrently with other behaviors within an ECU network.
- incorporating an ‘encapsulation shell’ which not only hides the implementation from other classes, but also prevents the implementation from directly (and incidentally) accessing the services of neighboring classes, in other words, it hides its implementation in both directions.
- establishing truly distributable objects. Since all clients and all servers are only interconnected by interfaces, communication even between remote objects is possible.

One of the interesting features of process classes is that they can have multiple interfaces, called SAPs (Service Access Points, for example in Figure 2 SwitchSAP, ClampSAP and DevSAP). Each interface represents a different aspect of a process class. Several collaborators can access different interfaces – even in parallel. SAPs are described in Section 2.3.

A process class uses its SAPs for all interaction with its environment. Communication is done by using method calls, aggregated to so-called protocols (see Section 2.4). Like a class in UML, a process class can have attributes forming its Process Data Set (PDS). Every SAP of a process class has at least one port.

2.3 SAPs

As mentioned above, a process class is encapsulated from its environment by SAPs. SAPs offering services to other classes are called server SAPs (like SwitchSAP and ClampSAP in Figure 2), whereas services required by the process class itself can only be requested through so-called client SAPs (like DEVSAP in Figure 2). SAPs offer or request services via ports. Ports are necessary to navigate method calls to its neighbors. Having a port of a client SAP connected to more than one class, it depends on the communication mode whether all connected classes will receive a request or only dedicated. Two communication modes are possible:

- peer-to-peer, meaning that the port has to be selected separately or,
- broadcast, where all connected classes will receive a request.

Since ports are associated with a SAP they can be interpreted as instances of a SAP. Vice versa, the SAPs are the type of the ports.
2.4 Protocols

Protocols, which are well known in telecommunications, are a set of messages two communicating entities must agree upon. This concept can be extended to a set of methods being called by objects. In TITUS, methods are functions that are offered by process classes to their neighbors at SAPs. The aggregation of all methods, offered at a server SAP or being incorporated at a client SAP, is called protocol. Protocols can be structured hierarchically using a single inheritance mechanism. A "NULL" protocol containing no methods represents the root of the protocol tree. The functionality of a protocol can be extended by creating a child protocol and adding new methods.

Figure 3 shows the protocol gen2state_c. Its parent is the NULL protocol having no methods whereas the gen2state_c protocol consists of the methods state0() and state1().

2.5 Finite State Machines

Finite State Machines (FSM) are the preferred means to describe the behavior of process classes. A finite state machine consists of an input vector X, a state Z and an output vector Y. In the methods of the process class, the boolean expressions over the PDS-elements and method arguments are evaluated to an associated element of the X-vector. Then, the state machine is 'started'. i.e., the X vector is evaluated and the appropriate transition is chosen. The transition sets the new state and manipulates the output vector Y. Then, the Y-vector is evaluated which might lead to the following actions: Manipulate the PDS variables, call a method of another process class, or manipulate the X-vector. The latter manipulation might seem at least unusual, but allows to call 'own services' in a transparent and efficient way. This procedure is repeated as long as there is a change in the X-vector. It is emphasized that this is one possible interpretation of finite state machines.
2.6 Frames

Frames allow hierarchical system design. They can include process classes or further frames as members and offer services at SAPs. Frames can use services of other frames or process classes on SAPs in client mode.

The top level frame of a system describes the entire structure of the application. Since in automotive software resources are always allocated statically and dynamic instantiation is not used, all port connections are already resolved at compile time.

Figure 4: A Frame containing a Function

2.7 Kinds of process classes

Process classes can be distinguished into client/server and firmware classes. Firmware classes are directly connected to sensors and actuators. Their behavior can be described by means of C-code. Since firmware classes directly incorporate HW-drivers, they are bound to specific ECUs. Client/server classes are independent from hardware and can be assigned to arbitrary ECUs in a mapping step described later. An exception are client/server classes next to firmware classes. These client/server classes are specialized into primary clients and primary servers respectively. All other client/server classes are called monitors.

2.8 OSEK Remote Procedure Call

Communication between process classes is asynchronous and explicit. To have control over the timing behavior of two process classes residing on remote ECUs it is necessary for the designer to be aware of the traffic a remote procedure call will generate. For example, a getValue() procedure call has to be modeled in TITUS with no return value. Instead of simply calling the getValue method of a (tentative stateless) server and expecting the result at a later point in time, the getValue() method can only set a flag at the server. The server will notice the set flag and calls the putValue(real result) method of the client with the result as parameter, i.e., the roles will change.

This none-stateless interpretation of a remote procedure call under automotive constraints, hence OSEK Remote Procedure Call, not only makes the timing implications explicit to the designer but furthermore encourages a clear design based on pure interfaces. Process classes support this design methodology.

OSEK is a standard for automotive embedded operating systems
3 Bringing the design to the vehicle

The functionality of a vehicle’s body electronics can be captured by several frames aggregating process classes. To make the application executable on an ECU network, every involved process class has to be mapped to an ECU [Figure 5].

![Figure 5: Mapping of Process Classes to ECUs](image)

Each process class instance also has to be mapped to OS tasks running on that ECU. The hierarchical structure of frames is resolved to allow the distribution of process classes. This development step is called flattening. Flattening produces a structure of process class instances and its connection topology. This structure is called software system. In a mapping step every process class instance is assigned to an ECU and an OS task. Connections between process class instances running on the same ECU are assigned to common variables and remote connections are mapped to CAN messages.

Automatic code generation provides executable code for a whole ECU – not only for the application function. This is code tiny enough to meet all constraints of series production ECUs.

4 Summary

The TITUS methodology supports the graphical programming of ECU software by tailoring concepts to the requirements of embedded automotive control software. The availability of commercial tools combining several production proven technologies will considerably shorten the design cycle of distributed, embedded automotive software.

References


