DEVELOPMENT OF DISTRIBUTED AUTOMOTIVE SOFTWARE
The DaVinci Methodology

Dr. Uwe Honekamp, Matthias Wernicke
Vector Informatik GmbH, Dep. PND - Tools for Networks and distributed Systems

Abstract: The software complexity in modern vehicle electronic systems is increasingly growing. Vehicle projects have to take into account a growing number of interconnected functions, which are jointly developed by many persons in many different companies. For facing these challenges, new design methodologies for a formalized and partially automated software development are required.

The DaVinci design methodology has been developed to match the specific requirements of distributed automotive systems. This includes the function-oriented design of the system structure as well as the deployment on a network and software integration on ECUs (electronic control units). Such a design serves as basis for an automatic code generation process, which integrates the applications into an efficient ECU target architecture with real-time operating system (RTOS) and communication stack.

Typical scenarios during the development processes like the reuse, exchange and integration of design data are supported and combined with a flexible configuration management. PC-based test environments may be used for functional integration tests or verification of the network communication.

This article is supposed to give a brief overview of the methodology as well as some selected aspects of its implementation in the DaVinci tool suite.

Key words: Design methodology, distributed embedded systems, automotive.

1. INTRODUCTION

The software architecture of typical automotive systems in general is required to fulfil important requirements, the most prominent being:

- *Independence of software component from each others:* it must be feasible to independently develop and test self-contained functions of the automotive software. Only by this means it is possible to **reuse** particular software components among model families.
- *Independence from particular network topology:* it must be possible to **map** a particular collection of communicating software components to a variety of network topologies.
- *Efficiency:* the **overhead** imposed by the conformance to a standard software architecture is required to be as **small as possible.** Of course, the optimal solution would be not to generate any overhead at all.

As a response to these requirements the DaVinci methodology has been developed. DaVinci supports the independence of "pieces of software" from each others by defining the so-called *software component.*

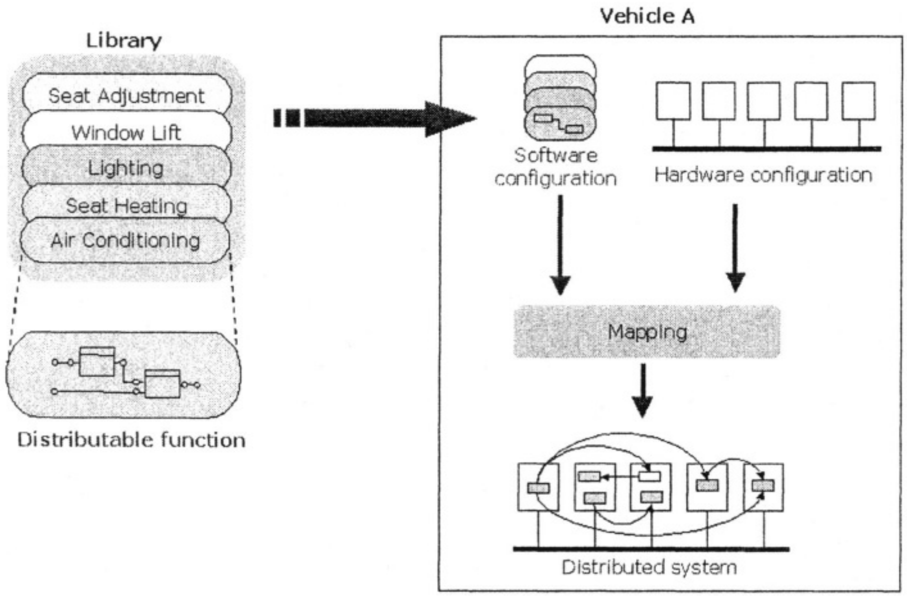

Figure 1. Distributed System

The combination of particular *software components* to form a logically consistent higher functionality is called a software configuration. In other words: the software configuration represents a graph of *software components.*

Furthermore, it is possible to model relevant hardware components (i.e. ECUs, communication buses, sensors, actuators). By this means a graph of hardware elements is formed.

In order to define a concrete network of *electronic control units* (ECU) for a particular car model the software graph has to be mapped on the hardware graph. The result is called a *mapping system.* The latter is the basis for sophisticated code generation activities that mainly resolve the abstract interfaces defined in the *software component.* By this means the strict interfaces leave virtually no overhead.

The persistent storage of DaVinci model elements is based on XML. Model elements that belong together are arranged in a so-called workspace. The latter, on the other hand, can be associated to a *configuration management* repository to support team-based development of DaVinci models.

The main field of application for DaVinci is the specification of so-called body electronics (e.g. power windows, climate control, seat adjustment, lighting, etc.). It is planned to extend the applicability of the DaVinci methodology to other automotive domains such as power train, chassis, etc. in the future.

2. DAVINCI DESIGN ELEMENTS

2.1 Software Component

Obviously, the main point for achieving independence of *software components* is the definition of strict interfaces among *software components* as well as to the underlying hardware and standard services (RTOS, communication drivers, network management, etc.).

A further point to take into account is the granularity of software gathered in a *software component.* The latter is defined as the minimum self-contained reusable software unit.

Software components can be arranged hierarchically such that *software components* (without behavioural description) contain other *software components.*

The behaviour of a *software component* can be modelled either directly by means of the C language or by means of some behaviour modelling tool such as The Mathwork's Simulink™ or I-Logix' Statemate™.

Furthermore, dSPACE's TargetLink™ can be used for behavioural modelling to support the creation of series production code from Simulink™ models.

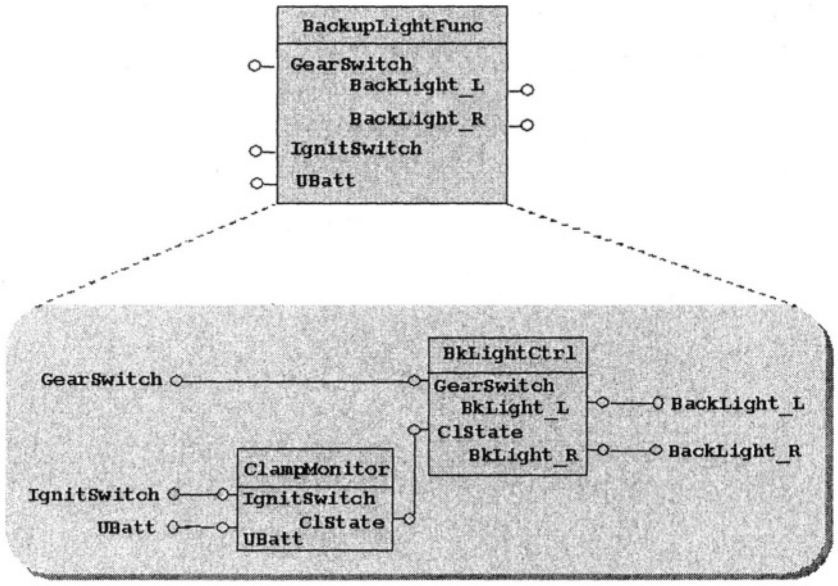

Figure 2. Example of a hierarchical software component

The architecture of Vector's DaVinci Tool Suite provides a generic adapter concept that allows to easily integrate additional behaviour modelling tools on customer's demand.

2.2 Signal

A signal is used for interconnecting *software components* with each others. Furthermore, a signal can be used to connect a *software system* (see ch. 2.4) to hardware devices such as *sensors* and *actuators.*

Signals maintain several properties such as the data type, the conversion formula from the physical domain to the data type as well as several automotive-specific properties like a timeout value.

2.3 ECU State Machine

The so-called *ECU state machine* is part of the abstraction mechanisms specially introduced to yield a maximum hardware independence of *software* *components. ECU state machines* define special states in which an ECU can operate as well as the possible transitions between states.

Software components, on the other hand, may define so-called *procedures* that implement special behaviour of the *software component* according to the current state of the *ECU state machine.*

Of course, the full power of *ECU state machines* as a means of abstraction is provided only if all ECUs (to which a particular *software component* can be mapped) implement an instance of the same *ECU state machine.*

2.4 Software System

A *software system* is a collection of *software components* connected by *signals.* The purpose of a *software system* is to gather *software components* to form a higher functionality based on the interaction of the particular *software components.*

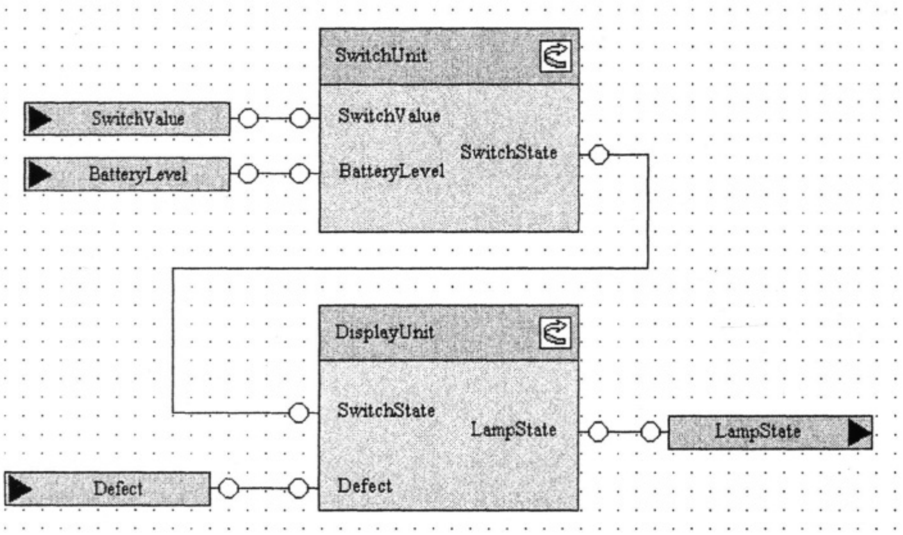

Figure 3. Example of a simple software system.

Another perspective of a *software system* is the representation of the entire (as far as DaVinci in concerned) software functionality of the network of ECUs in a car.

3. DAVINCI IMPLEMENTATION ELEMENTS

3.1 BUS

A *bus* is used to express a communication device among the collection of ECUs in a network. In general, several types of buses (e.g. CAN, LIN, MOST, FlexRay, etc.) could be used in a network of automotive ECUs. Furthermore, several disjoint buses of the same type (e.g. CAN) could be used to form subnets connected to each others by gateway ECUs.

DaVinci currently supports the definition of CAN networks. Gateways are supported but the gateway functionality must be explicitly specified by a *software component.*

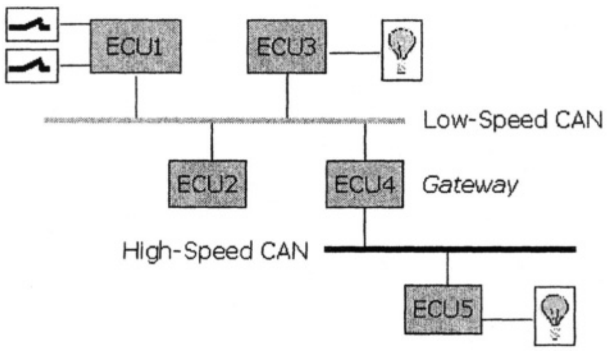

Figure 4. Communication Bus

For the future, DaVinci will not only support a wide range of buses but it will be possible to automatically determine the communication behaviour of gateway ECUs based on the formal description of the communication network.

3.2 ECU

An *ECU* obviously is used to carry out computations, i.e. host a collection of *software components* as described by the mapping description. An *ECU* can have *sensors* and *actuators* that resolve particular signals.

In other words: *software components* interact (via *signals*) with the real world by means of *sensors* and *actuators* provided by the *ECU* onto which the *software component* is mapped.

Furthermore, the description of an *ECU* consists of a reference to a particular *ECU state machine.*

For supporting the generation of target code, it is possible to specify the type and (if applicable) variant of the underlying microcontroller.

3.3 Hardware System

A *hardware system* is the direct counterpart of the concept of the *software system,* i.e. it describes a collection of *ECUs* as well as the communication *buses* used to interconnect particular *ECUs.*

3.4 Mapping System

A *mapping system* defines a particular combination of a *hardware system* and a *software system.* Furthermore, the mapping of *software components* to *ECUs* as well as the mapping of *signals* to bus messages are essential parts of the description of a *mapping system.*

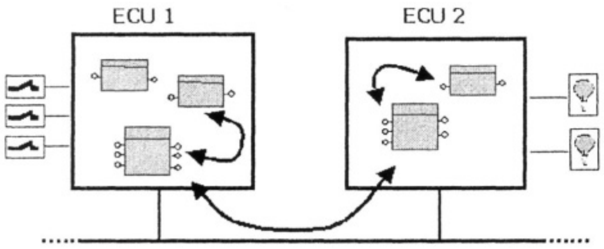

Figure 5. A simple mapping system

4. DAVINCI TARGET ARCHITECTURE

As mentioned before, the abstract interfaces defined by the DaVinci methodology at some point in time must be resolved such that *software components* can be embedded into a defined target architecture with maximum efficiency.

DaVinci's code integration capabilities allow the seamless integration of heterogeneous *software components,* e.g. legacy C Code and model-based developed components.

The standardized target architecture used for DaVinci models is depicted in *Figure 6.* As sketched by the picture, the target architecture itself consists of self-contained modules some of which must be specially generated according to the DaVinci model configuration.

Other modules exist as predefined source code but must be configured (i.e. header files containing configuration information must be created). Both generation and configuration of software modules is carried out by specially tailored code generation tools.

One of the most prominent code generation issues is the generation of a communication stack that provides signal access in the context of a so-called *interaction layer* to *software components.*

The underlying RTOS must be configured as well. The configuration depends on the mapping of *software components* as well as the assignment of priorities to tasks.

Furthermore, timing constraints concerning bus communication must also be taken into account, i.e. it must be made sure that messages with a high priority are send by tasks that as well are executed under a high priority.

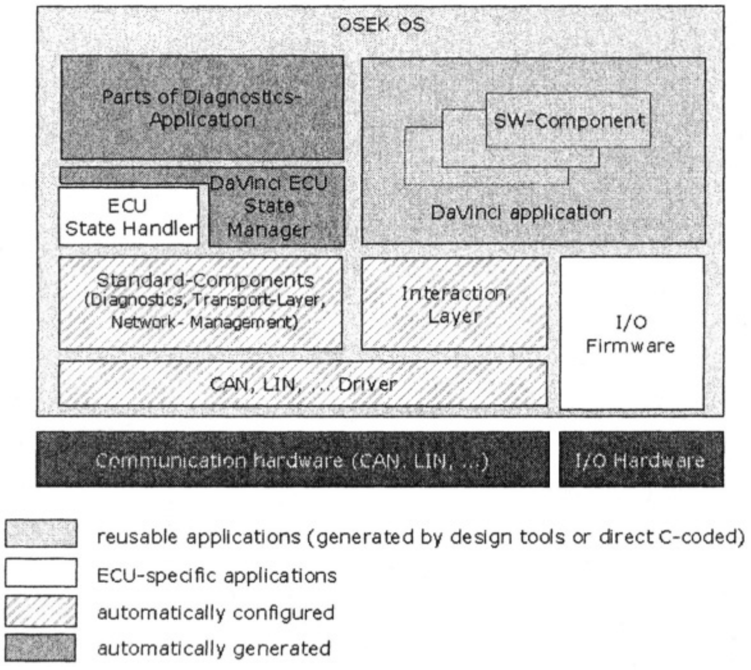
OSEK OS
Parts of Diagnostics-Application
SW-Component
DaVinci ECU State Manager
ECU State Handler
DaVinci application
Standard-Components (Diagnostics, Transport-Layer, Network- Management)
Interaction Layer
I/O Firmware
CAN, LIN, ... Driver
Communication hardware (CAN, LIN, ...)
I/O Hardware
reusable applications (generated by design tools or direct C-coded)
ECU-specific applications
automatically configured
automatically generated

Figure 6. DaVinci target architecture

The currently supported range of microcontroller targets consists of the Motorola PowerPC series for embedded applications as well as on the Star 12 microcontroller. Further targets are supported on demand.

For diagnostics purposes Vector's CANdesc software module can be integrated to DaVinci applications. Supported interaction layers are

GMLAN, DBKom, and derivatives of the Vector IL. For calibration and measurement purposes it is possible to add a CCP driver as well.

5. TEST OF SOFTWARE COMPONENTS

A very typical constraint of the development of ECU software is the fact that in early phases of the development project the target hardware does not yet exist.

This fact should not have an impact on the activities on the software side. For this purpose it is essential to have a test platform that emulates especially the communication hardware.

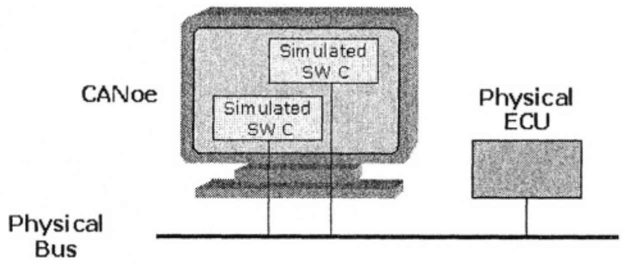

Figure 7. CANoe test environment

This requirement is fulfilled by the software tool CANoe that has as well been developed by Vector Informatik GmbH. The functionality of CANoe, however, is not limited to mere simulation.

It is possible to simulate particular nodes of a communication bus that otherwise already exists in hardware, i.e. physically existing ECUs can be conveniently combined with simulated ones and perform realistic real-time communication among each others. This concept is depicted by *Figure 7.*

DaVinci supports CANoe as an experimentation platform for early phases of a network project. It is possible (by means of a predefined target configuration) to generate the target architecture code such that the entire ECU including applications, communication stack and RTOS can be simulated by CANoe.

6. DISTRIBUTED DEVELOPMENT

Large software projects in the automotive domain are usually distributed among several supplier companies under the control of the car manufacturer.

A tool suite for carrying out large software project must therefore support the distributed development of software.

For this purpose Da Vinci provides several features:

- It is possible to attach a DaVinci workspace to a configuration management repository thus enabling DaVinci to be used by the development team of at least an entire company.
- DaVinci provides sophisticated import and export mechanisms that are capable of dealing with the formal model without uncovering subjects to intellectual property (e.g. the structure of a control algorithm implemented to realize a specific *software component*). This technique can be used to share DaVinci model elements among developers of different companies. Please consult *Figure 8* for more details.

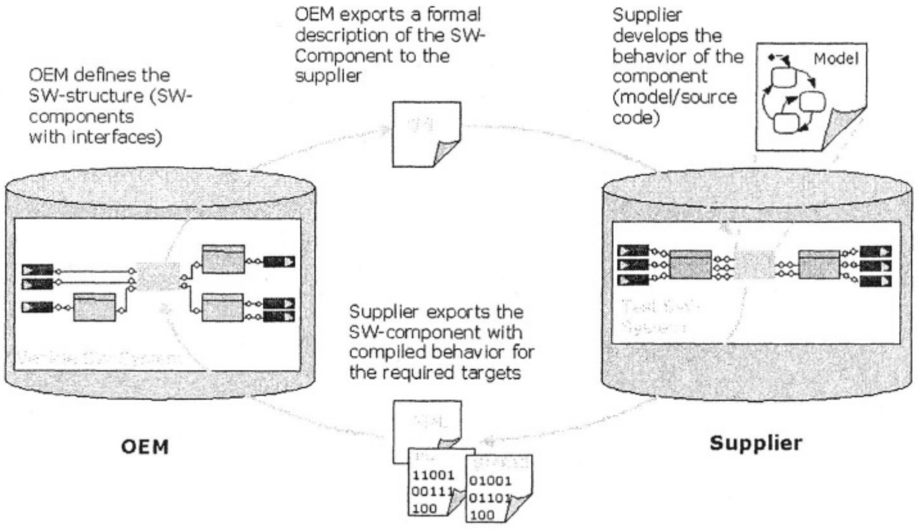
OEM exports a formal description of the SW-Component to the supplier
Supplier develops the behavior of the component (model/source code)
Model
OEM defines the SW-structure (SW-components with interfaces)
Supplier exports the SW-component with compiled behavior for the required targets
OEM
Supplier
11001
00111
100
01001
01101
100

Figure 8. Distributed development using DaVinci

7. FUTURE PROSPECTS

The DaVinci methodology as well as the corresponding tool suite is subject to continuous improvement. Among other things this applies in particular to the introduction of further behaviour modelling tools as well as the support for additional microcontroller targets.