ABSTRACT
This paper presents a distributed software architecture for safety-related embedded systems in modern automobiles. The safety architecture ensures that software and hardware faults are treated correctly and consistently by a distributed system that consists of various control units connected by a bus. This is especially important for safety-related vehicle applications.

The general design of the architecture as well as its concrete implementation will be described. It will be shown how the abilities of the OSEK operating system and its standardized components are used to seamlessly integrate distributed control units in such a way that the behavior of the different individual control units ensures the safety of the vehicle.

INTRODUCTION
The general structure of the functional software is similar in most vehicle applications. The functional software is built on top of basic software modules that offer some kind of run-time environment and interfaces to the hardware. The basic software modules are application-independent. There are committees that try to standardize these software modules.

STRUCTURE OF THE FUNCTIONAL SOFTWARE
The functional software in an embedded control unit can be decomposed into three main execution modules: Input, Control and Output. Figure 1 shows the information flow through such an embedded system.

If a functionality is implemented as a distributed system that involves various control units, the execution modules Input and Output not only communicate with sensors and actuators that are directly connected to the control unit, but also with other control units via a bus. A bus that is widely used in automobiles is controller area network (CAN), defined by ISO 11898.

The execution modules of a system run cyclically. In an electronic throttle control system, for example, the execution module Control calculates a new throttle position every 12.5ms and the execution module Output, which generates a pulse-width modulated signal for the direct-current motor that moves the throttle, runs every 2ms [1]. In this example the execution modules require different cycle times, and very often they must be executed with different priorities. To ensure this, a run-time environment is necessary that organizes the different execution modules into tasks.

STANDARD SOFTWARE MODULES
Standard software modules for distributed control units in vehicles have been defined in a joint project between German and French automotive companies. The industry standard is called OSEK1 /VDX2 and comprises the following three areas:

- Operating System: OSEK OS3 is a real-time, multitasking operating system for the software in electronic control units and forms the basis for the other modules [2].
- Communication: OSEK COM4 is a software module that facilitates data exchange between tasks within a control unit as well as between different control units [3].
- Network Management: OSEK NM5 is a software module that determinates and monitors the network configuration [4].
The standard software modules have been designed to avoid high and recurring expenses for the development of software modules that are not application-related. Another important factor was the goal to achieve compatibility of control units even if they are made by different manufacturers. This goal has been achieved by standardizing interfaces and protocols.

OSEK OS constitutes the basis for the integration of automotive applications made by various manufacturers by standardizing the API \(^6\) of the operating system. It ensures the portability and reusability of software modules and hides the hardware-specific implementation details. In addition, the operating system ensures that all real-time requirements of the different applications can be met. The API of OSEK OS covers the following areas:

- **Task management:** Interfaces to activate, terminate and switch tasks.
- **Resource management:** Interfaces to control the access of various software modules to jointly used data and program memory sections.
- **Event management:** Interfaces to synchronize activities and tasks.
- **Interrupt management:** Interfaces that can be used for interrupt processing.
- **Error management:** Interfaces that standardize error handling.
- **Alarm management:** Interfaces to handle relative, absolute, static and dynamic alarms.

OSEK COM implements the general communication functionality. It offers interfaces to transfer data between tasks and/or interrupt service routines. OSEK COM hides the differences between internal and external communication. This means that for the application programmer it is transparent whether the communication takes place between tasks that reside in the same control unit (internal communication) or whether different control units are involved (external communication).

OSEK NM provides services related to networks of communicating control units. It coordinates global operation modes, such as for example the network start-up or a network wide sleep mode to save energy. It also helps to monitor the operating states of the different network nodes.

**CONCEPT OF THE SAFETY ARCHITECTURE**

The standardized interfaces of the OSEK/VDX software modules allow the specification of a distributed safety architecture that is independent from the underlying hardware and that can be used to coordinate the applications of various suppliers. The safety architecture is built on top of the standard software modules and uses the application program interfaces offered by OSEK OS, OSEK COM and OSEK NM.

**SAFETY ARCHITECTURE MODULES**

The safety architecture structures the safety-related software, which is required in addition to the functional software, to ensure that possible failures are detected and handled correctly. The safety-related software consists of three types of modules: monitors, safety interface modules and safety control modules [5]. Figure 2 shows the information flow between the modules of the safety-related software and the functional software.

Figure 2 Safety architecture

Monitors are used to detect failures of hardware and software components. Embedded systems in vehicles normally consist of many sub-components. To detect all possible failures it is necessary to implement a huge number of monitors, which are executed in the tasks and interrupts of the control unit. Some of them are executed cyclically every x milli-seconds, while others are only activated in certain situations.

Safety interface modules are used to represent the hardware and software components that are important to ensure the safety of the system. They determine and trace the current status of the components by analyzing the fault notifications of the monitors used to check the component. If a new, safety-related component is added to a system, new monitors and a new safety interface module have to be implemented.

The main objective of the safety interface modules is to concentrate the information gathered by the monitors. They form a level of abstraction such that the safety control modules do not have to care about individual fault notifications, but get the required information – the component status – directly.

Safety interface modules are specified as state machines (see Figure 5). The active state of the state machine corresponds to the current state of the component represented. A safety interface module changes its internal state if one of its input monitors generates a fault notification.

---

\(^6\) Application Program Interface
Safety control modules ensure that only correctly working components are used and degrade the functional software if a required component fails. Every execution module has its dedicated safety control module that uses the status information of the required components to determine how the execution module has to fulfill its function. The purpose for separating the execution modules that implement the system functionality from the safety control modules which control safety-related activities is to separate the functional software from the safety-related software.

Similar to the safety interface modules, the safety control modules are also specified as state machines. A safety control module changes its state if it is notified by a safety interface module about a failure of a component that is relevant for the controlled execution module.

**DESIGN OF THE SAFETY ARCHITECTURE**

The conceptual idea behind the safety architecture can be applied to different kinds of embedded systems. In the following it will be explained how the safety architecture can be implemented on communicating control units that use OSEK OS as the operating system. It will be shown how the API of OSEK COM facilitates the realization of a distributed safety architecture.

**COLLABORATION BETWEEN THE MODULES**

Figure 3 shows a collaboration diagram of the safety architecture. Each module of the safety architecture is represented as an object and the information flow is realized with the help of synchronous and asynchronous method calls between the objects.

Monitors are objects that implement methods to check the correctness of hardware or software components. If a monitor detects a failure it calls the synchronous method `handleNewFault` of the safety interface module representing the component. The method `handleNewFault` has the input parameter `monitorID`, which is a unique identifier of the calling monitor. In the method the safety interface module analyzes the identifier and changes its state if necessary. Then the method returns.

Normally the state machines of the safety interface modules are simple (no hierarchy) and the time needed to process a new fault notification is short. Therefore using a synchronous method call is possible – even if the calling monitor is running inside an interrupt service routine. Synchronous method calls have the advantage that their implementation is simple and efficient, because the use of communication services is not necessary.

Various monitors that may run concurrently in different tasks or interrupt service routines can call the method `handleNewFault` of a safety interface module. Therefore it is necessary to protect the code of the method as a critical section – e.g. by using the OSEK resource mechanism.

Every safety control module is implemented as a separate high-priority task. Tasks are used to protect the memory and stack sections of the safety control modules. The high task priority ensures that the safety control modules are executed in time. Compared to the safety interface modules, the safety control modules are more complex, and the processing of the method `handleNewState` can take some time. Therefore this method is called asynchronously to avoid that the calling safety interface module – and indirectly the calling monitor – has to wait until the method has been processed. Message queues provided by OSEK COM are used to implement the asynchronous method calls.

In situations, in which no new failures are detected and the component states remain unchanged, the safety control modules wait, without requiring any run time. If a monitor detects an error, it calls the method `handleNewFault` of its safety interface module. The safety interface module changes its active state and sends the OSEK message `handleNewState` to the interested safety control modules. The safety control modules that receive the message are automatically woken up and process the new state.

The main advantage of using OSEK message queues is that they facilitate the communication between different control units. This is necessary if a safety control module is interested in the state of a remote component (see Figure 4). A disadvantage of the OSEK message queue mechanism is that the messages are always processed in first-in-first-out order. Messages have no priority.

![Figure 3 Collaboration diagram](image1)

![Figure 4 Remote safety interface module](image2)
module checks its execution mode. Then it fulfills its function according to the current mode. The safety control module of an execution module sets its current execution mode through the method `setMode`. OSEK offers two mechanisms how a safety control module can define the mode of an execution module that is running in a different task. The safety control module can send an un-queued message to the execution module or it can write the new mode into a global memory section protected by a resource. The message has the advantage that the safety control module does not depend on the correct resource handling of the execution module.

IMPLEMENTATION OF THE SAFETY INTERFACE AND SAFETY CONTROL MODULES

The safety interface and safety control modules are implemented as state machines. There are various well-known approaches how the state machines can be implemented:

1. Nested switch
2. State event table
3. State machine class

The safety interface modules can be implemented easily by using the nested switch or the state event table approach. For the more complicated safety control modules that are specified as hierarchical state machines the state machine class is used. In the following sections examples for the different implementations will be given and the advantages and disadvantages of the different approaches will be discussed.

Nested switch

Figure 5 shows a graphical representation of the state machine of a safety interface module that represents a pressure sensor (PS). The C code that implements this state machine with the help of a nested switch construct is shown in Figure 6.

```
void cPS_handleNewFault(cPS *me, unsigned const id) {
    switch (me->state) {
    case NO_ERR:
        switch (id) {
            case a:
                handleNewState(a);
                break;
            case b:
                handleNewState(b);
                break;
            case c:
                handleNewState(c);
                break;
        }
        break;
    case TEMP_ERR:
        break;
    case ERROR:
        break;
    } ...
}
```

Figure 6 C implementation of the nested switch approach

The main advantage of the nested switch approach is that it requires only the memory for a scalar state variable. A disadvantage of the approach is that the time needed to process a fault notification is not constant. The time depends on the switch branch in which the fault is processed. Another disadvantage is that the C code cannot be reused. Every state machine needs its own nested switch-case constructs.

State event table

In the state event table approach the state machine is transformed into a table that is stored in the permanent memory of the control unit (see Figure 7). Every table cell contains a function pointer that defines which actions must be executed and a constant that indicates the next state.

```
static Trans const cPS_Table[MAX_STATE][MAX_SIG] = {
    {{ handleNewState_a, TEMP_ERR },
     { DoNothing, NO_ERR },
     { handleNewState_b, ERROR }},
    {{ DoNothing, TEMP_ERR },
     { handleNewState_c, NO_ERR },
     { DoNothing, ERROR }},
    {{ DoNothing, ERROR },
     { DoNothing, ERROR },
     { DoNothing, ERROR }};
};
```

Figure 7 State event table

The function `handleNewFault` (see Figure 8) is generic and can be used for all safety interface modules. Only the tables depend on the actual structure of the state machine.

```
void handleNewFault(cSIM *me, unsigned const id) {
    Trans const * = me->table + me->state + me->Signals + id;
    if (!(*-action)(me)) {
        me->state = -nextState;
    }
}
```

Figure 8 Generic `handleNewFault` function
Advantages of this approach are that the method handleNewFault can be reused and that the time needed to process the method is constant. Disadvantages are that the approach requires more memory to store the state event table and that – to avoid bugs – tool support should be used to modify the topology of the state machines. If for example the safety interface module of the pressure sensor must handle a new fault notification, three new lines must be added to the table shown in Figure 7. Another disadvantage is that it is necessary to implement a function for every transition action.

The implementation of hierarchical state machines by using state event tables is difficult. With the nested switch approach hierarchical state machines can be implemented by nesting more switches, but for complex hierarchical state machine the code gets chaotic. With both approaches entry and exit actions must be transformed into transition actions. This may cause additional code because the actions that belong to a state must be implemented for every transition entering or leaving the state again.

State machine class

For the specification of the safety control modules both hierarchy as well as entry and exit actions are used. Therefore the state machine class approach is used to implement the safety control modules. Every safety control module is derived from a generic class called Statemachine (Figure 9). Safety interface modules call the method handleNewState to inform the safety control module about a changed component state. The states of a safety control module are represented as methods implemented by the derived safety control module. In the attribute myState of the generic class Statemachine a function pointer to the method that represents the currently active state is stored. The method handleNewState (Figure 10) executes the function that is stored in myState.

Figure 9 State machine class

The advantage of the state machine class approach is that hierarchical state machines can be implemented easily by encapsulating the complexity of handling the hierarchy into the generic Statemachine class [6]. Another advantage is the reusability of the Statemachine class.

RESULTS

The state machines of the safety interface and safety control modules are normally defined with graphical tools such as Stateflow from MathWorks. The code generators integrated into these tools enable persons that are unfamiliar with C programming to generate C code from the specification. Writing the C code manually requires more time, and it is hard to avoid bugs, even if it is clearly defined how the state machines and all additional features like parallel and hierarchical state machines or history junction have to be implemented.

After analyzing the different possibilities of how the software modules of the safety architecture may communicate and how the safety interface and safety control modules can be implemented, an adequate code generator must be found. There are many code generators that very often use the nested switch approach to generate code from the specified state machines, but none of them can generate code that can be directly integrated into a distributed safety architecture that consists of software modules running on different control units. Until now it is necessary to manually write wrappers around the generated code, because the code generators offer no support for communication and distribution. This may change with the new version of the tool TargetLink (dSPACE) that will have a close integration with OSEK OS [7].

However, before any code generator can be used for the generation of safety-related software a general requirement must be fulfilled. A code generator for safety-related systems must either have a certificate of validation to a recognized national or international standard or it must be assessed to establish its fitness for its purpose (IEC61508 Part 3 Section 7.4.4.3a).

REFERENCES

   http://www.osek-vdx.org
   http://www.osek-vdx.org
4. OSEK/VDX OSEK Network Management Concept and Application Programming Interface (Version 2.5.2).
   http://www.osek-vdx.org
   http://www.osek-vdx.org
   http://www.dspace.de

**CONTACT**

Melanie Cossy is a system designer at STZ Softwaretechnik (Germany) and provides consulting services related to the development of embedded systems. Starting with the software design for a digital signal processor, Melanie has been engaged in a wide range of development activities in the recent years – examples are requirement management and test automation processes, the integration of standard components and the development of software architectures. She earned her MSc in *Distributed Computing Systems* at the Brunel University (West London) where she designed an Extensible Framework for testing Distributed Object-Oriented Systems. At present Melanie develops software safety architectures for safety-related embedded control systems on behalf of a large automotive company. This work is part of her dissertation research activities at the Eberhard-Karls University in Tübingen, Germany. Melanie has presented at the SAE 2003 World Congress in Detroit and the DESIGN&ELEKTRONIK Conference in Germany, and has written for Automotive Electronics magazine.

e-mail address: melanie.cossy@stz-softwaretechnik.de
web address: www.cossy.de/melanie.