Formal Verification of Interacting Software Components

Dipl. Ing. (FH) Melanie Cossy, MSc
STZ Softwaretechnik
Im Gaugenmaier 20
73730 Esslingen
Germany
melanie.cossy@stz-softwaretechnik.de
www.stz-softwaretechnik.de

Abstract: This paper presents a software architecture for embedded control systems, which allows the application of formal verification methods to ensure that the interaction between different software components does not violate important system properties. The paper starts with an introduction of the component types, their interfaces and the data and control flows of the software architecture. The functionality of the different component types is introduced and it is explained how their internal logic can be specified formally, by modeling finite-state machines. After the formal specification of the individual software components a model checker can be applied to formally verify that their behavior is correct and satisfies important system properties. The paper shows how the important system properties can be specified formally and how a model checker can be used to prove mathematically the required properties. The paper concludes with a discussion of the advantages achieved through the application of formal methods.

Motivation

In the last years an increasing number of embedded control systems has been applied in safety-related application domains – for example in the avionics or automotive sector. The software of safety-related embedded control systems must fulfill functional as well as safety requirements. Safety requirements specify for example self-checking capabilities that determine dangerous hardware and software failures, or failure handling mechanisms that passivate defective components and activate alternative means of computation. Functional software requirements are for example the processing of sensor input-signals and the regulation of actors.

In safety-related application domains some functional requirements are so important that they must remain available to ensure system safety. Therefore the application functions that realize these functional requirements must be able to fulfill their functionality despite of hardware and software faults. If a required resource fails and is not available anymore, alternative hardware or software resources must be used. The operation mode of an application function defines which hardware and software resources can be used and how the application function must realize its function. Table 1 shows six exemplary operation modes of an application function, which determines a current pedal position with the help of three different travel sensors.
<table>
<thead>
<tr>
<th>operation mode</th>
<th>short description</th>
<th>required hardware resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode=0</td>
<td>functionality not available</td>
<td>-</td>
</tr>
<tr>
<td>mode=1</td>
<td>determination of the current pedal position: high resolution</td>
<td>travel sensor 1, 2, 3</td>
</tr>
<tr>
<td>mode=2</td>
<td>determination of the current pedal position: high resolution</td>
<td>travel sensor 1, 2</td>
</tr>
<tr>
<td>mode=3</td>
<td>determination of the current pedal position: high resolution</td>
<td>travel sensor 1, 3</td>
</tr>
<tr>
<td>mode=4</td>
<td>determination of the current pedal position: low resolution</td>
<td>travel sensor 2, 3</td>
</tr>
<tr>
<td>mode=5</td>
<td>determination of the current pedal position: low resolution</td>
<td>travel sensor 3</td>
</tr>
</tbody>
</table>

Table 1 Operation modes of an application function

It is extremely important that the operation modes of the different safety-related application functions are always adequate for the current situation. Therefore dedicated software components are used to implement the safety logic that determines the operation modes.

Software components of the safety logic

Figure 1 shows how the safety logic can be structured into software components with clearly defined tasks and interfaces. The main objective of this design is to ensure that the complex safety logic is decomposed into components that can be specified and tested thoroughly.

In the following paragraphs the different functions of the safety logic are introduced and it is explained how these functions collaborate:

**Monitor functions** are responsible for the detection of component failures and the generation of fault notifications to inform the safety logic, which then can select an appropriate failure handling strategy. Which strategy is appropriate depends on the nature of the generated fault notification. Some fault notifications indicate permanent component failures, other fault notifications inform about recoverable failures or about an abnormality that cannot be localized directly (fault type and location is unknown). The first kind of fault notification can be handled directly by passivating the affected component. The second kind of fault notification requires special temporary safety measures. An example for such a temporary safety measure is an error identification strategy that gets necessary after an ambiguous fault notification. During the error identification routine the operation modes of the application functions are changed to check the possible failure sources separately.
**Component status functions** track the current status of the safety-related hardware and software components. Every component status function represents a real-world component and combines the fault notifications of the monitor functions, which check this component, to calculate its component status. Figure 2 shows how the behavior of a component status function that represents a travel sensor can be specified. The component status function has three input values: fault notification 1 (sensor signal in the error range), fault notification 2 (sensor signal rises too fast) and fault notification 3 (incorrect communication protocol). If any fault notification gets true a state transition from the state *in service* to the state *out of service* is triggered and in dependence of the currently active state the output value "status travel sensor" is set to *inService* or *outOfService*.

![Component Status Function](image)

**Figure 2 Component status function of a travel sensor**

Every application function has its own dedicated **reconfiguration control function** that analyzes the component status values calculated by the component status functions to determine an operation mode that passivates the faulty components. Figure 3 shows the specification of a reconfiguration control function, which determines the operation modes of an application function that determines a current pedal position with the help of three different travel sensors. The different states of the reconfiguration control function represent the different operation modes of the application function (Table 1) and transitions between these states take place if a required travel sensor fails.

![Reconfiguration Control Function](image)

**Figure 3 Reconfiguration control function**

The functions, described so far, are able to handle fault notifications that can be handled directly. Ambiguous fault notifications, which do not indicate the exact fault location and type, as
well as fault notifications, which indicate a failure that perhaps can be eliminated during normal operation, must be handled differently by executing a temporary error recovery or error identification strategy.

Error recovery and error identification strategies typically involve more than one application function. Therefore a **reconfiguration coordination function** is necessary to coordinate the execution of such a strategy. The reconfiguration coordination function informs the reconfiguration control functions about the currently active temporary measures. It handles the priorities of the different temporary safety measures and considers all relevant system states that may influence the decision which temporary measures should be activated or deactivated.

Table 2 shows four exemplary temporary measures and Table 3 contains the application functions that must be coordinated to execute these temporary measures.

<table>
<thead>
<tr>
<th>temporary measure</th>
<th>priority</th>
<th>start condition</th>
<th>stop condition</th>
<th>simultaneity/exclusion</th>
<th>interruptible</th>
<th>operation mode combination during activation</th>
<th>operation modes that interrupt/inhibit the measure</th>
<th>maximal number of interrupted executions</th>
<th>maximal number of successful executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Check of the pressure sensors</td>
<td>1</td>
<td>fault notification: &quot;hydraulic leakage &quot; (FN_a)</td>
<td>after C1max time steps</td>
<td>exclusive execution</td>
<td>yes</td>
<td>X: 4, Y: 5, Z: 4, W: 1</td>
<td>X: 2</td>
<td>1</td>
<td>infinite</td>
</tr>
<tr>
<td>2: Localization of a hydraulic leakage</td>
<td>2</td>
<td>always after C1 time steps</td>
<td>after C2max time steps</td>
<td>exclusive execution</td>
<td>yes</td>
<td>X: 1, Y: 5, Z: 1</td>
<td>X: 2, Y: 2, Z: 2</td>
<td>2</td>
<td>infinite</td>
</tr>
<tr>
<td>3: Check of the pedal sensors</td>
<td>3</td>
<td>always after C1 time steps</td>
<td>notification: &quot;sensor signals available&quot; (FN_c)</td>
<td>simultaneously with measure 4</td>
<td>no</td>
<td>X: 5, Y: 5, Z: 1</td>
<td>X: 2</td>
<td>3</td>
<td>infinite</td>
</tr>
<tr>
<td>4: Recovery from a contamination in the hydraulic circuit</td>
<td>4</td>
<td>fault notification: &quot;contamination&quot; (FN_b)</td>
<td>notification: &quot;fault recovery successful&quot; FN_d</td>
<td>simultaneously with measure 3</td>
<td>yes</td>
<td>Y: 5, Z: 3</td>
<td>Y: 2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2 Temporary safety measures

<table>
<thead>
<tr>
<th>application function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>Determination of the brake pedal position</td>
</tr>
<tr>
<td>X</td>
<td>Calculation of the required braking force</td>
</tr>
<tr>
<td>Y</td>
<td>Regulation of the braking force</td>
</tr>
<tr>
<td>Z</td>
<td>Control of the hydraulic pressure supply</td>
</tr>
</tbody>
</table>

Table 3 Application functions

The state machine that specifies the behavior of a reconfiguration coordination function, which can coordinate the four measures from Table 2, is shown in Figure 4. The four parallel state machines in the upper part of Figure 4 (request1, request2, request3 and request4) track whether a temporary measure is requested or not. If certain conditions inhibit the execution of a temporary measure permanently, a state transition to the state off takes place. Below the four state machines there is another state machine called tempMeasurement that contains six states representing the different possible combinations of active temporary measures (Table 4).

The state machine tempMeasurement determines which temporary measure combination must be executed. The activation and deactivation conditions depend on the variables req1, req2, req3 and req4 that are set from the state machines request1, request2, request3 and request4.
Verification of the specification

The state machines of the component status functions, the reconfiguration control functions and the reconfiguration coordination function are specified with a graphical CASE tool like Statemate or Stateflow for example. The advantage of these CASE tools is that the specification is executable and can be used during testing. This significantly minimizes the necessary test effort. But if the specification is used to decide about the correctness of the implementation it is important to ensure that the specification itself is correct.

Manual review techniques are the traditional approach to ensure the correctness of specifications. The problem of manual reviews is their reliance on human intuition and the huge effort needed to execute reviews rigorously. To minimize the required time, manpower and costs caused by reviews, the traditional approach can be extended with formal verification activities to ensure that the specification possesses different general and application specific properties.
There are various general properties that every specification, which uses state machines to define the required system behavior, must possess. Examples for general properties are reachability and determinism. Reachability means that every state must be reachable and every transition condition must be enabled at least in one situation. Determinism requires that the state machines behave deterministically. This implies that all state transitions that leave the same state must be disjunct (in every moment only one state transition is enabled).

Apart from the general properties there are application specific properties. Criticality assessments of the application domain help to identify which system properties should be verified formally. For the presented architecture the main objective of the formal verification is to ensure the interaction between the different components and the coordination of temporary safety measures.

The following list contains six system properties that are important for the correct handling of temporary measures:

- **System property 1**: It must be ensured that only certain temporary measures are active simultaneously.
- **System property 2**: If a temporary measure is activated the involved reconfiguration control functions must react and switch their application functions into an appropriate operation mode. If the operation mode combination is incorrect the temporary measure must be interrupted.
- **System property 3**: The priorities of the temporary measures must be handled correctly.
- **System property 4**: Temporary measures that have a defined maximal execution interval must be terminated in time.
- **System property 5**: It must be ensured that a temporary measure is not executed more times than allowed.
- **System property 6**: Temporary measures with a maximal number of interruptions must be deactivated if the maximal number is reached.

**Applied Formal Method**

The safety logic is specified with the help of state machines and a well-known method to formally verify state-based systems is **model checking**. Model checkers implement efficient algorithms that automatically traverse the modeled system to check the defined system properties for all possible input combinations.

Figure 5 visualizes the different elements of a tool support that can be used to formally verify the specified state machines with a model checker. The graphically specified state machines are converted into a textual representation that the selected model checker can process. To facilitate the definition of the required system properties the tool offers a graphical user interface and automatically generates a formal representation of the defined properties.

The model checker uses the formal specification of the safety logic and the required system properties to prove mathematically that the safety logic possesses the defined properties. If the verification fails, two different possible causes must be analyzed: the modeled state machines may be wrong or the specified properties may be too strict. An expert must evaluate the failed property and decide, whether the specification of the safety logic or the defined properties must be changed. This means that the specification phase and the verification phase cannot be separated strictly. They must be performed alternately until all properties are satisfied.
Figure 5 Formal verification

Figure 6 Graphical user interface
Formal Specification of System Properties

Figure 6 illustrates how a normal user can easily specify the required system properties. With the information defined by the user in the graphical user interface the tool support must generate corresponding formal expressions:

**System property 1:** It must be ensured that only certain temporary measures are active simultaneously.

The CTL expression below defines that if the temporary measures 1 is active (COOR_.tM[0]=1) no other temporary measures are allowed to be active simultaneously.

\[
\text{SPEC AG (COOR_.tM[0]=1} \rightarrow \neg (\text{COOR_.tM[1]=1} \lor \text{COOR_.tM[2]=1} \lor \text{COOR_.tM[3]=1})\]

Similar expressions can be specified for the temporary measures 2, 3 and 4. The only difference is that the temporary measures 3 and 4 can be executed simultaneously.

\[
\begin{align*}
\text{SPEC AG (COOR_.tM[1]=1} & \rightarrow \neg (\text{COOR_.tM[0]=1} \lor \text{COOR_.tM[2]=1} \lor \text{COOR_.tM[3]=1}) \\
\text{SPEC AG (COOR_.tM[2]=1} & \rightarrow \neg (\text{COOR_.tM[0]=1} \lor \text{COOR_.tM[1]=1}) \\
\text{SPEC AG (COOR_.tM[3]=1} & \rightarrow \neg (\text{COOR_.tM[0]=1} \lor \text{COOR_.tM[1]=1})
\end{align*}
\]

**System property 2:** If a temporary measure is activated the involved reconfiguration control functions must react and switch their application functions into an appropriate operation mode. If the operation mode combination is incorrect the temporary measure must be interrupted.

The following CTL expressions define that if the reconfiguration coordination function activates a temporary measure now, then in the next time step the involved reconfiguration control functions must select appropriate operation modes or another time step later the temporary measure may not be requested any more.

\[
\begin{align*}
\text{SPEC AG(COOR_.tM[0]=1} & \rightarrow \text{AX}(((\text{SCM_W_.BA}=1 \lor \text{SCM_W_.BA}=2) \land \text{SCM_X_.BA}=4 \land \text{SCM_Y_.BA}=5 \land \text{SCM_Z_.BA}=4) \lor \text{AX} !\text{COOR_.tM[0]=1}) \\
\text{SPEC AG(COOR_.tM[1]=1} & \rightarrow \text{AX}(((\text{SCM_X_.BA}=5 \land \text{SCM_Y_.BA}=5 \land \text{SCM_Z_.BA}=5) \land \text{AX} !\text{COOR_.tM[1]=1}) \\
\text{SPEC AG(COOR_.tM[2]=1} & \rightarrow \text{AX}(((\text{SCM_W_.BA}=4 \land \text{SCM_X_.BA}=3) \land \text{AX} !\text{COOR_.tM[2]=1}) \\
\text{SPEC AG(COOR_.tM[3]=1} & \rightarrow \text{AX}(((\text{SCM_Y_.BA}=5 \land \text{SCM_Z_.BA}=3) \land \text{AX} !\text{COOR_.tM[3]=1})
\end{align*}
\]

**System property 3:** The priorities of the temporary measures must be handled correctly.

The temporary measure 1 has the highest priority and if this measure is requested it must be directly executed. A request for the temporary measure 1 is only delayed, if the temporary measure 3 is active, because measure 3 cannot be interrupted. This behavior is defined by the following CTL expression.

\[
\text{SPEC AG((COOR_.request1=active} \land \text{COOR_.tM[2]=0} \rightarrow \text{AX(COOR_.tM[0]=1} \land \neg \text{COOR_.request1=active})
\]

For temporary measure 2 a corresponding expression can be specified. If a request for temporary measure 2 (second highest priority) is active and the measure 1 is not requested and the temporary measure 3 is not active, then temporary measure 2 must be executed in the next step.

\[
\text{SPEC AG((COOR_.request2=active} \land \text{COOR_.tM[2]=0} \land \neg \text{COOR_.request1=active} \rightarrow \text{AX(COOR_.tM[1]=1} \land \neg \text{COOR_.request2=active})
\]

Similar CTL expressions can be defined for the measures 3 and 4:

\[
\begin{align*}
\text{SPEC AG((COOR_.request3=active} \land \neg \text{COOR_.request1=active} \land \neg \text{COOR_.request2=active} \rightarrow \text{AX(COOR_.tM[2]=1} \land \neg \text{COOR_.request3=active}) \\
\text{SPEC AG((COOR_.request4=active} \land \neg \text{COOR_.request1=active} \land \neg \text{COOR_.request2=active} \rightarrow \text{AX(COOR_.tM[3]=1} \land \neg \text{COOR_.request4=active})
\end{align*}
\]
System property 4: **Temporary measures that have a defined maximal execution interval must be terminated in time.**

Some temporary measures are only allowed to be active for a limited time period. In the example from Table 2 the time interval of the temporary measure 1 is limited. With the help of a counter that is incremented every time step when the state tempM1 is active, the CTL expression below specifies that the time interval of temporary measure 1 must be shorter than a defined maximum duration time.

\[ \text{SPEC } \text{AG}(\text{COOR\_tM}[0]=1 \rightarrow \text{COOR\_c1}<\text{COOR\_C1max}+2) \]

System property 5: **It must be ensured that a temporary measure is not executed more times than allowed.**

It is forbidden to execute the temporary measure 2 infinitely often. Therefore the counter suc2 is incremented every time the temporary measure 2 is executed successfully and a CTL expression is used to specify that the temporary measure 2 can be executed only 3 times.

\[ \text{SPEC } \text{AG}(\text{COOR\_suc2}<4) \]

System property 6: **Temporary measures with a maximal number of interruptions must be deactivated if the maximal number is reached.**

The variable err1 is used to count the interruptions of temporary measure 1. The first interruption of the temporary measure 1 causes its permanent deactivation. This behavior is checked with the following CTL expression:

\[ \text{SPEC } \text{AG}(\text{COOR\_err1}<2) \]

### Conclusion

During the development of safety-related embedded control systems many test series are required to ensure that the behavior of the embedded control system is correct. Using state machines to specify the safety logic has the advantage that such a specification form can be simulated and executed. Simulation helps to gain a good understanding of the specified
behavior and the possibility to execute the specification facilitates test automation. This reduces the required test effort significantly, because with an executable specification the implementation can be tested directly against the specification: the implementation and the specification are executed with the same input values and the resulting output of the implementation is checked against the output generated by the executable specification. But to achieve reliable test results it must be ensured that the executable specification is correct.

Traditionally manual reviews are used to verify specification documents. But for complex systems conventional review techniques are inefficient and error prone, because the quality of the verification results relays on human intuition and judgement. This problem can be solved by applying formal verification techniques to prove automatically and very efficiently that the specification possesses important system properties. The advantages of formal methods are particularly large if changes or adaptations of the specification are frequent. Figure 7 shows for example that with the model checker SMV only a few seconds are needed to verify all the systems properties that have been explained before.

References

- M. Cossy, Formale Spezifikation der Sicherheitssoftware für die Rekonfiguration von elektronischen Systemen im Fahrzeug, Safetronic, 2003
- M. Cossy, Software-Sicherheitsarchitektur für Steuergerate im Fahrzeug, Entwicklerforum Kfz-Elektronik, 2003