Model-based Generation of Run-time Monitors for AUTOSAR

Lars Patzina[1], Sven Patzina[1], Thorsten Piper[2], and Paul Manns[2]

[1] Real-Time Systems Lab, Technische Universität Darmstadt, Darmstadt, Germany
{sven.patzina, lars.patzina}@es.tu-darmstadt.de
[2] DEEDS Group, Technische Universität Darmstadt, Darmstadt, Germany
{piper, manns}@cs.tu-darmstadt.de

Abstract. Driven by technical innovation, embedded systems, especially in vehicles, are becoming increasingly interconnected and, consequently, have to be secured against failures and threats from the outside world. One approach to improve the fault tolerance and resilience of a system is run-time monitoring. AUTOSAR, the emerging standard for automotive software systems, specifies several run-time monitoring mechanisms at the watchdog and OS level that are neither intended, nor able to support complex run-time monitoring. This paper addresses the general challenges involved in the development and integration of a model-based generation process of complex run-time security and safety monitors. A previously published model-based development process for run-time monitors based on a special kind of Petri nets is enhanced and tailored to fit seamlessly into the AUTOSAR development process. In our evaluation, we show that efficient monitors for AUTOSAR can be directly modeled and generated from the corresponding AUTOSAR system model.

Keywords: AUTOSAR, extended live sequence charts, model-based, monitor petri nets, run-time monitoring, signatures

1 Introduction

Embedded systems are becoming increasingly interconnected and can no longer be considered as being separated from the outside world. A prominent example are multimedia systems in the automotive domain that are connected to the internet without being totally separated from safety-critical components. Many systems have been developed as closed systems and often little attention has been paid to security mechanisms such as encryption and safe component design to deal with errors and attacks. Even modern networks in the automotive domain are vulnerable to passive and active attacks [7] and protocols such as the CAN bus protocol have been identified as a major security drawback [9]. This makes it necessary to secure safety-critical components and their communication, even if they are not directly accessible. To secure these systems, Papadimitratos et al. [13] propose a secured communication and demand a secure architecture.

However, in the majority of cases, even such efforts cannot eliminate all security vulnerabilities that can lead to safety threats as it is impossible to foresee all attacks during development. Moreover, it is often economically or technically infeasible to secure existing systems retroactively against external adversaries. Hence, systems and especially electronic control units in cars cannot be considered as secure against attacks, either caused by unknown vulnerabilities or by the required integration of legacy components. To cope with these security and safety issues, run-time monitoring is a feasible approach to detect attacks that exploit previously unknown errors and security vulnerabilities [11].

The AUTOSAR (AUTomotive Open System ARchitecture) platform[3] is emerging in the automotive domain as an open industry standard for the development of in-vehicular systems. To cope with the growing complexity of modern vehicular systems, it provides a modular software architecture with standardized interfaces and a run-time environment (RTE) that separates application level software components from the underlying basic software modules and the actual hardware. AUTOSAR offers a clearly structured system specification, which is stored in the standardized AUTOSAR XML (ARXML) format. The AUTOSAR development process supports the monitoring of control flow and timing properties at a low abstraction level [1, 2], but does not provide support for modeling complex monitoring functionality at the software component level.

For this purpose, we have adopted our generic Model-based Security/Safety Monitor (MBSecMon) development tool chain [15]. It is based on the Model-Driven Development (MDD) concept that supports the generation of monitors from signatures describing the interactions between the components of a system. The MBSecMon specification language (MBSecMonSL) is based on Live Sequence Charts (LSC) introduced by Damm and Harel [5], which have been extended [15] for the modeling of behavioral signatures. A specification based on these extended LSCs (eLSC) and a structural description of the system constitutes the input set of the MBSecMon process. The signatures are divided in intended system behavior (use cases) and known attack patterns and attack classes (misuse cases). These signatures are automatically transformed to a formally defined Petri net language, the Monitor Petri nets (MPNs) [14], which serve as a more explicit, intermediate representation. With the provision of platform-specific information, security/safety run-time monitors are automatically generated for different target platforms.

The contribution of this paper is the development of a methodology for the model-based development of complex run-time monitors for AUTOSAR that operate directly on the AUTOSAR system model. We depict the challenges that arise during the development and integration of a model-based monitor generation framework into the AUTOSAR development process and present solutions to each. In summary, these challenges are as follows.

C1 Integrating existing AUTOSAR development fragments into the monitor generation process.

[3] AUTOSAR: http://www.autosar.org

C2 Providing type safety during the whole modeling and generation process of the monitors.
C3 Modeling monitor signatures on the same abstraction level as the AUTOSAR system models.
C4 Mapping of the abstract signature descriptions to platform specific monitoring code.
C5 Providing communication data of the generated AUTOSAR software components (SW-C) to the monitors.
C6 Supporting the relocatability of software components by generating distributed monitors from global signatures.
C7 Generating monitors with a minimal overhead for the control units.

The remainder of the paper is structured as follows: Section 2 gives an overview of existing approaches for monitoring and instrumentation and describes how they can be applied in the AUTOSAR process. In Sect. 3, the challenges are described in detail, and solutions are presented based on the adaptation of the MBSecMon process for the AUTOSAR development process. The generated monitors that are connected to the example system are evaluated in Sect. 4. In Sect. 5, a conclusion is drawn, and possible future work is discussed.

2 Related Work

The AUTOSAR standard specifies several run-time monitoring mechanisms that are provided by the Operating System (OS) and the Watchdog Manager (WdM). In the *Specification of Operating System* [1], three monitoring mechanisms for the detection of timing faults at run-time, i.e., tasks or interrupts missing their deadline, are considered. *Execution time protection* monitors the execution budget of tasks and category 2 Interrupt Service Routines (ISRs), in order to guarantee that the execution time does not exceed a statically configured upper bound. *Locking time protection* supervises the time that a task or category 2 ISR can hold a resource, including the suspension time of OS interrupts or all interrupts of the system. This is done to prevent priority inversions and to recover from potential deadlocks. The third monitoring mechanism is *inter-arrival time protection*, which controls the time between successive activations of tasks and the arrival of category 2 ISRs. These mechanisms monitor the system at the task level and are not suited to implement control flow or data flow monitoring. The configuration is done at the OS level and does not factor the system view of the model.

The *Specification of Watchdog Manager* [2] provides three monitoring mechanisms that are complementary to those offered by the AUTOSAR OS. All of the implemented mechanisms are based on checkpoints that report to the watchdog manager (WdM) when they are reached. Supervised functions have to be instrumented with calls to the watchdog, which verifies at run-time the correct transition between two checkpoints as well as the timing of the checkpoint transitions. For *alive supervision*, the WdM periodically checks if the checkpoints of a supervised entity have been reached within the given limits, in order to detect

if a supervised entity is run too frequently or too rarely. For the supervision of aperiodic or episodical events that have individual constraints on the timing between checkpoints, the WdM offers *deadline supervision*. In this approach, the WdM checks if the execution time of a given block of a supervised entity is within the configured minimum and maximum bound. The third mechanism that the WdM provides is *logical monitoring*, which focuses on the detection of control flow errors, which occur if one or more program instructions are processed either in an incorrect sequence or are not processed at all. This approach resembles the work of Oh et al. [12], in which a graph representation of a function is constructed by dividing the function into *basic blocks* at each (conditional) branch. Basic blocks are represented by nodes, whereas legal branches are represented by arcs that connect the nodes. Whenever a new basic block is entered, the monitor verifies that the taken branch was legal, according to the graph representation.

Except for control flow monitoring, the monitoring mechanisms that AUTOSAR currently specifies are only suitable for monitoring timing properties at a low level of abstraction. None of the approaches is configured on the level of the AUTOSAR system model, which offers the developer an intuitive and integrated view on the system. Apart from the monitoring services offered by AUTOSAR, few research has covered this area so far. One exception are two articles by Cotard et al. [3, 4], in which the authors describe a monitoring approach for the synchronization of tasks on multi-core hardware platforms. In their approach, dependencies between tasks are first modeled as a finite state machine (FSM). The FSM is then translated into a linear temporal logic (LTL) specification, from which Moore machines and, subsequently, C code is generated. Their primary focus is on synchronization and not on control or data flow monitoring.

In our MBSecMon process, an extended version of the expressive and compact Live Sequence Charts is used as signature specification language. Kumar et al. [10] specifies protocols with LSCs and transforms them for verification to temporal logic formulas. Thereby, complex LSC specifications lead to an explosion of the formula and are, therefore, with LSCs as specification language not suitable as intermediate language for the code generation for embedded systems.

Besides these transformations to LTL, there are some approaches that use Petri nets directly for the specification of monitors. Frankowiak et al. [6] uses Petri nets to specify low cost process monitors on a micro controller. He enhances regular Petri nets by token generators and special end places (bins). Additionally, subnets are linked by a control net. For complex monitor specifications, these nets get relatively large compared to MPNs, whose semantics include an implicit evaluation logic when an event does not match to the specified sequence. The MPNs are tailored to describe monitor specifications in an explicit but nevertheless compact form.

3 The MBSecMon Framework

The Model-based Security/Safety Monitor (MBSecMon) Framework [15] has been developed as a generic approach to build tool chains for monitor generation.

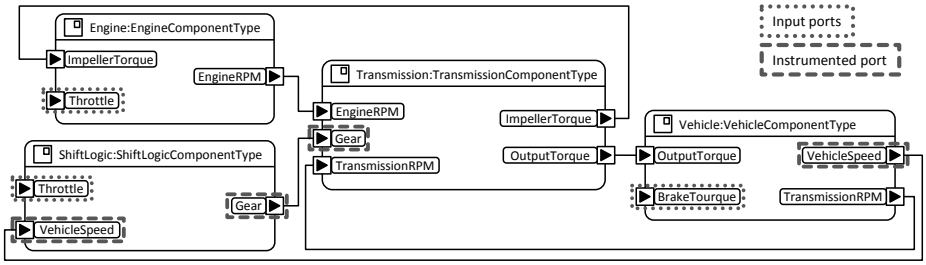

Fig. 1: AUTOSAR example model of an automatic transmission

In this section, we show how this framework has been tailored to fit seamlessly in a model-based AUTOSAR development process and solve the challenges that are raised in Sect. 1. This overcomes the current lack of model-based monitoring support in AUTOSAR tool chains that we attested in Sect. 2.

3.1 Example: Automatic Transmission Controller

The example used throughout this paper focuses on monitoring the communication between AUTOSAR software components (SW-Cs). The AUTOSAR system is modeled with the tool OptXware Embedded Architect[4] and the implementation of the subsystems is generated by the AUTOSAR code generation of MathWork's Embedded Coder plugin for Simulink.

The implementation of the example system is based on the *Automatic Transition Controller* demo project [18], shipped with Matlab/Simulink. As depicted in the system model in Fig. 1, this model describes the internal behavior of three application-level software components, *ShiftLogic*, *Transmission*, and *Engine*, based on the input values *Throttle* and *BrakeTorque* and the interaction between these components. Influences such as aerodynamics and drag friction of the wheels are represented by the *Vehicle* block. To comply with the needs of the AUTOSAR code generation provided by Embedded Coder, the behavioral model has been adapted by replacing all continuous with discrete blocks. The generated code serves as the behavioral implementation of the skeleton generated by the AUTOSAR tool OptXware Embedded Architect. For the asynchronous communication between the components, the sender-receiver communication pattern is used.

3.2 The Tailored Monitor Generation Process for AUTOSAR

The generic MBSecMon development tool chain has been extended to enable a seamless integration into the AUTOSAR development process. Figure 2 depicts on the left side the original simplified AUTOSAR development process, starting

[4] OptXware: http://www.optxware.com

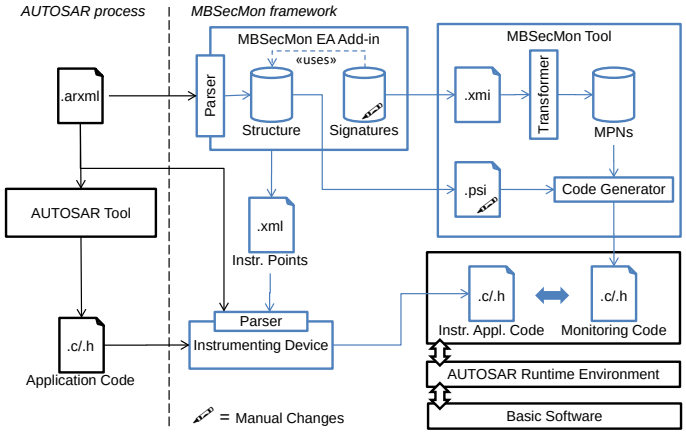

Fig. 2: MBSecMon framework embedded in AUTOSAR development process

with the system structure description persisted in the AUTOSAR XML format (ARXML). This file is used by the AUTOSAR tool chain to generate the RTE and a code skeleton for the implementation of the SW-Cs, which is supplied by the Simulink code generator. The final code is usable in an AUTOSAR simulation environment or directly on the Electronic Control Unit (ECU) of a vehicle.

On the right-hand side of Fig. 2, the framework is depicted that embeds the monitor generation tool chain in the AUTOSAR development process. The specification of monitor signatures is achieved with the help of a tailored version of the UML2 modeling tool Enterprise Architect[5] (EA). This tool has been extended by an add-in that allows the modeling of eLSCs (*Signatures*) depending on the imported component diagram view (*Structure*) of the AUTOSAR system.The modeled signatures are then exported together with additional platform specific information (PSI) extracted from the imported AUTOSAR model. Additionally, the *MBSecMon add-in* analyses the modeled signatures for needed *instrumentation points* in the AUTOSAR application code and persists this information in an XML file. Through a graph-based model-to-model transformation [16], the exported representations of the signatures are transformed in the formally defined Monitor Petri nets (MPNs) [14] that serve as an intermediate language used for a straight-forward code generation. The code generator translates the signatures, represented as MPNs, incorporating the additional information (PSI), to monitoring code. This monitor is stimulated by calls of its interfaces. Therefore, the *instrumenting device* uses the AUTOSAR system specification, together with the *instrumentation points* file, to instrument the application code. The instrumentation is realized via interface wrappers, which intercept the communication between two components. A detailed description of the instrumentation of the interfaces of AUTOSAR components with wrappers is given in [17].

[5] SparxSystems Enterprise Architect: http://www.sparxsystems.com

This process allows for specifying and automatically generating monitors on an abstract level based on the system information provided by the AUTOSAR development process. Only the specification for the monitors has to be modeled using the MBSecMon add-in and the platform specific information has to be extended by the system engineer. In the following, based on the example in Sect. 3.1, this process and the necessary adaptation based on the identified challenges are described in detail.

Challenge 1: Integrating existing development fragments. In the AUTOSAR development process, the system structure is modeled at a high abstraction level, describing the software components (SW-C), their ports with specified data types, and the connections between the ports. The generated monitors should observe the communication between these SW-Cs. Therefore, a component view of the AUTOSAR system should be used to support the modeling of signatures.

MBSecMon process: The MBSecMon EA add-in incorporates an ARXML parser that imports the AUTOSAR software component structure, which has been modeled in an AUTOSAR system-level editing tool, such as OptXware Embedded Architect. Derived from this data, the add-in creates an UML component diagram as shown in Fig. 3 that includes the components, the ports and a connector representation based on the AUTOSAR naming scheme. Additional system information such as the names of the connectors and ports are stored as tagged values in the model elements and are abbreviated in the diagram view for a clear presentation.

Example: The software components in Fig. 3 comply to the blocks of the Simulink model used to generate the implementation of the application code. This model is used in the MBSecMon development process to describe the allowed and forbidden communication sequences between the components as signatures.

Challenge 2: Providing type safety. In the AUTOSAR development process and in the automotive domain, specialized data types are used. This allows for limiting the value range of these data types and prevents wrong assignments such as storing a speed value in a variable intended for the impeller torque. Generated monitors have to obey this implicit safety mechanism that is inherent in the AUTOSAR standard.

MBSecMon process: The special data types provided by the AUTOSAR system specification are imported along with the component diagrams and stored in its ports. While modeling the signatures, the developer's choice is constrained to these data types. These are further used in the monitor generation process and result in type safe monitoring code.

Example: In the ARXML file the data type for the vehicle speed, shown in Listing 1.1, is defined as *VehicleSpeedDataType* with a base type *Double* and is limited to a range of possible values.

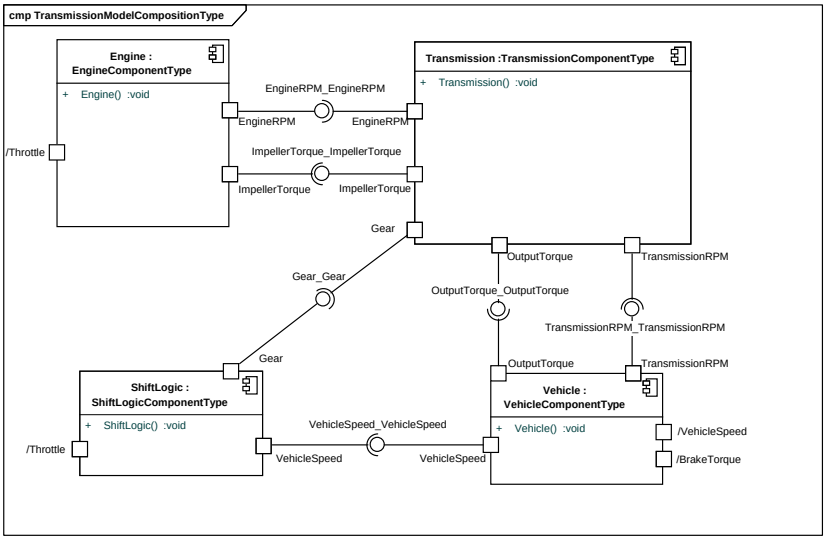

Fig. 3: UML component diagram of the system imported to EA

Listing 1.1: Type safety in AUTOSAR (ARXML file)

```
<REAL–TYPE >
  <SHORT–NAME>VehicleSpeedDataType</SHORT–NAME>
  <LOWER–LIMIT INTERVAL–TYPE="CLOSED" >−1000.0</LOWER–LIMIT>
  <UPPER–LIMIT INTERVAL–TYPE="CLOSED" >1000.0</UPPER–LIMIT>
  <ALLOW–NAN>false</ALLOW–NAN>
  <ENCODING>DOUBLE</ENCODING>
</REAL–TYPE>
```

Challenge 3: Modeling at the same abstraction level. AUTOSAR system specifications are modeled at a high abstraction level as presented in Challenge 1. The ports describe which type of communication is used to interact with other SW-Cs over the Virtual Function Bus (VFB). Thus, it makes sense to monitor the communication between the SW-Cs. A widespread approach to describe interactions between components are various kinds of sequence charts that are on the same abstraction level as the AUTOSAR specification. These descriptions of the interaction between SW-Cs (signatures) can be used to generate monitors.

MBSecMon process: For this purpose, the MBSecMon specification language (MBSecMonSL), which consists of extended Live Sequence Charts (eLSC) that are structured by use/misuse cases (UC/MUC), is used in the MBSecMon framework. In addition to the concepts of the wide-spread Message Sequence Charts [8], eLSCs distinguish between hot (solid red border) and cold (dashed blue border) elements, where hot elements are mandatory and cold elements are optional. Furthermore, two forms of eLSCs exist, an universal eLSC with a prechart (precondition) (blue dashed hexagon) before the mainchart (solid black rectangle) and an existential eLSC without a precondition. In the prechart, every element

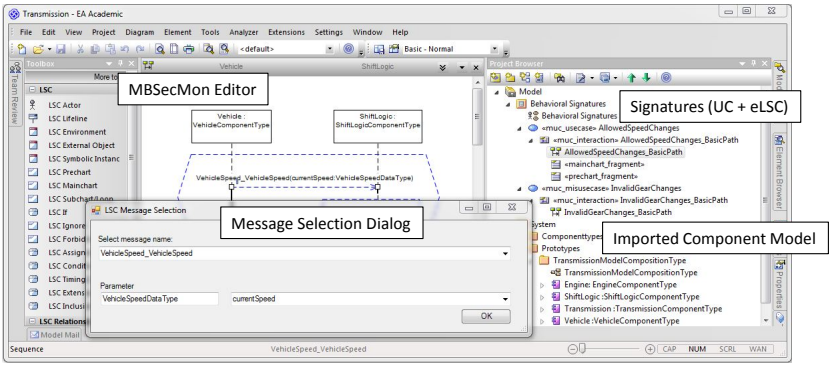

Fig. 4: Tailored Signature Editor for AUTOSAR

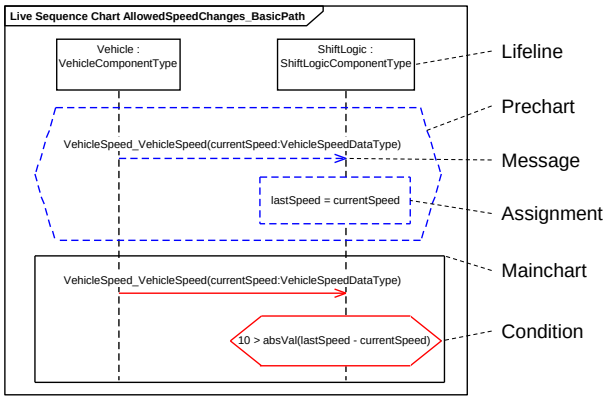

Fig. 5: Signature of the Use Case "AllowedSpeedChanges"

is interpreted as mandatory. This is based on the interpretation of the prechart as a precondition. When the prechart is fulfilled, the part of the signature in the mainchart is evaluated.

The import of the component view from an AUTOSAR tool to EA forms the basis for the signature modeling. The MBSecMon EA add-in (Fig. 4) provides a context sensitive choice of messages between components and their parameter types. This ensures the compliance to the modeled AUTOSAR system.

Example: Figure 5 shows a simple example of a concurrent signature that uses only the basic elements of the eLSC language. This signature monitors the communication between the *Vehicle* and the *ShiftLogic* components by initializing the monitor for every message, consisting of a sending and a receiving event, transmitted over the port *VehicleSpeed*. The message contains the value *currentSpeed* that is stored by the assignment to an eLSC specific variable *lastSpeed*. The first processed sending event triggers the initialization of a new instance of

Listing 1.2: LSC specific information in the PSI file

```
<entry key="AllowedSpeedChanges_BasicPath.context_method.absVal">fabs
</entry>
```

the signature that concurrently monitors the next messages. The next *VehicleSpeed* message is processed by the mainchart of the first instance and evaluated by the condition to the previous value stored in *lastSpeed*. This monitoring instance is then terminated based on the result of the condition. Subsequently, the same message is evaluated by the prechart of the second instance of the signature and overwrites the variable *lastSpeed* with the new value.

Challenge 4: Mapping to platform specific monitoring code. By modeling signatures on a much more abstract level than the code of the target platform and using an annotation language in the signatures, the code generator needs additional information about the mapping to the target platform.

MBSecMon process: To support type safe, AUTOSAR-compliant interfaces for the monitor, additional mappings are generated to the platform specific information (PSI) file. It contains mappings between AUTOSAR instrumentation interface and the internal events of the monitor, data types for transferred values, a code mapping of signature annotations to the target domain, and configuration details for the code generation. Most information for the file can be automatically derived from the imported system specification and the modeled signatures. Only mappings exported by the EA add-in from annotated pseudo code in the signatures have to be adapted manually to the target language. For more convenient usage, a mapping library for these annotations could automate this manual step.

Example: The generated PSI file provides the mapping of pseudo code used in the signatures to platform-conform code fragments, as depicted in Listing 1.2 for a method *absVal* in the condition of the signature in Fig. 5.

Challenge 5: Providing communication data to the monitors. The system model of the AUTOSAR specification uses the concept of the Virtual Function Bus (VFB) and only describes the communication pattern (e.g. sender/receiver). This communication is realized on the target platform depending on the RTE and the allocation of the SW-Cs to the different ECUs. This hampers the monitoring of the communication between the SW-Cs.

MBSecMon process: Based on the signatures, the *instrumenting device* needs a clear naming convention for the monitor interfaces to allow for the wrapping of write and read methods in the AUTOSAR system code. Additionally, the *instrumenting device* uses the exported information that specifies, based on the signatures, which ports in the AUTOSAR system have to be instrumented to minimize the footprint of the instrumentation by only adding method calls for the required events.

Listing 1.3: Monitor interfaces for the running example

```
void dispatchEvent_TransmissionModelCompositionType_Vehicle_
  VehicleSpeed(VehicleSpeedDataType*);
void dispatchEvent_TransmissionModelCompositionType_ShiftLogic_
  VehicleSpeed(VehicleSpeedDataType*);
```

Example: The interfaces of the monitor are named based on the naming conventions of the AUTOSAR standard. Thus, the *instrumenting device* can automatically generate calls to the interfaces into the system code. Listing 1.3 shows the interfaces of the generated example monitor. The method name is derived by concatenation of the string `dispatchEvent_` and the full name of the port of the AUTOSAR component and the data type of the parameter origins from the AUTOSAR model.

Challenge 6: Supporting the relocatability of software components. One of the important concepts of the AUTOSAR standard is the relocatability of SW-Cs that allows for distributing them to different ECUs without changing the specification. Due to modeling the monitors on the same abstraction level as the AUTOSAR system specification, the developer of the signatures cannot incorporate information about the final distribution of the SW-Cs. The code generation process must support the generation of distributed monitors based on the actual distribution of the SW-Cs. This reduces the run-time overhead for single ECUs and the communication overhead over the buses between the ECUs in contrast to a central monitor.

MBSecMon process: In the MBSecMon generation process, the signatures modeled as eLSCs already contain an affiliation of the events (sending and receiving) to the SW-Cs. This affiliation is obtained when the exported signatures are transformed to the intermediate Monitor Petri nets (MPN), and allow for generating distributed monitors based on the SW-C located on the same ECU. To preserve dependencies between the part monitors (e.g. sending before receiving event of a message), an additional communication between the monitors has to be established. Therefore, the code generator has been prepared to create identifiers that can be replaced by macro definitions to system communication methods.

Example: In the example, we use the sender-receiver pattern that directly supports transferability and exchange of AUTOSAR software components. The two components, *ShiftLogic* and *Vehicle*, in the signature in Fig. 5 can be distributed to different ECUs by defining "ShiftLogic;{Vehicle}" in the PSI file the instance that is on the same ECU (*ShiftLogic*) and the instances (*Vehicle*) that is located on another ECU and should be synchronized with it.

Challenge 7: Generating monitors with a minimal overhead. The generated run-time monitors are deployed on the ECUs and run alongside the SW-Cs.

Hence, their induced run-time and memory overhead on the ECUs have to be sufficiently small. The reasonable overhead depends on the required level of safety and security that has to be reached by monitoring. For the runtime-overhead, a worst case upper bound can be calculated by a static analysis of the signatures in the MPN format.

Example: In the presented example in Figure 5 the monitor has to evaluate two transitions per monitor instance (sending or receiving events) in one event processing step. Additionally, to this computation the annotated assignment or condition has to be taken into account.

The evaluation in Sect. 4 shows the overhead that the resulting monitors introduce to the system.

4 Evaluation

For the evaluation of the MBSecMon specification and generation process for AUTOSAR, an adaptation of the Simulink example model *Automatic Transmission Controller* [18], as presented in Sect. 3.1, is used. In order to provide compatibility with Simulink's AUTOSAR code generator, which does not support the continuous blocks (such as integrators) that were used in the original example, we decomposed the example model into separate software components and replaced the incompatible blocks with their corresponding discrete versions. The generated code contains the runnables (executable entities) of each software component (SW-C), which have to be integrated into the implementation code skeleton that was generated by the system level design and simulation tool OptXware Embedded Architect (OXEA). In OXEA, we have also designed the system model that corresponds to the Simulink example, and which is stored in the standardized ARXML format.

We instrumented the evaluation system with two different monitors (cf. Fig. 1). The first one is *AllowedSpeedChanges*, as presented in Sect. 3, which monitors the communication between the components *Vehicle* and *ShiftLogic* in order to detect a communication error between these components. The monitor signals an error in case that it detects a difference between two consecutive speed readings that is larger than 10 mph within a 20 ms timeframe (the period of the tasks). The second one is *InvalidGearChanges*, which, in contrast to the first signature, monitors the misuse case of a gear shifting by more than one step within a 20 ms timeframe. Therefore, the communication between the components *ShiftLogic* and *Transmission* is monitored. Based on these two signatures, a monitor that consists of a controller and a monitor representation for each signature is generated by the MBSecMon process.

The evaluation covers the run-time overhead that the monitoring induces per instrumented port (*Gear* for ShiftLogic and Transmission, and *VehicleSpeed* for Vehicle and ShiftLogic), and per instrumented runnable (Vehicle, ShiftLogic, Transmission). Furthermore, we analyze the memory overhead of the monitors, for both, code and data segments. Finally, a scalability analysis on an embedded system is performed.

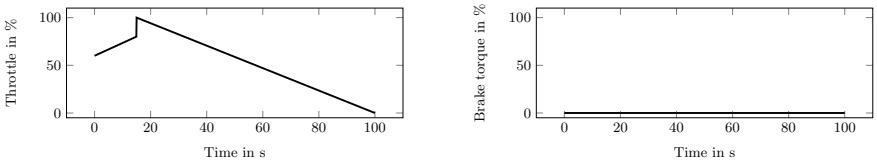

Fig. 6: Input data set for the passing maneuver

Table 1: Execution time comparison of original and monitored calls to the RTE

Component	RTE Call	Average *Original (ticks)*	*Instr. (ticks)*	*Diff. (%)*
ShiftLogic()	Rte_Read_VehicleSpeed ...	24,76	38,87	57
ShiftLogic()	Rte_Write_Gear ...	18,00	23,60	31
Transmission()	Rte_Read_Gear ...	25,28	37,63	49
Vehicle()	Rte_Write_VehicleSpeed ...	21,56	27,62	28

Run-time analysis. The evaluation was conducted using OptXware EA's simulation environment on a AMD Phenom II X4 955 processor, running at 3.20 GHz. The timing measurements were taken using the Win32 API functions *MyQueryPerformanceCounter* and *MyQueryPerformanceFrequency*, which are Window's high resolution timing functions, providing CPU tick granularity. This is a best effort solution, as there is no commonly agreed on reference architecture for such evaluation. Therefore, we also provide relative measurements of the execution time overhead as comparison.

Figure 6 shows the input data (readings of the throttle and brake torque sensor) for the test run of the automatic transmission model. It represents a passing maneuver, where the vehicle approaches a slower car and then abruptly accelerates to pass the car.

Table 1 shows the run-time for performing calls to the RTE in the simulation environment without (*Original*) and with instrumentation (*Instr.*). The measured time for the instrumented RTE calls includes the wrapper around the call method, the invocation of the monitor, and the evaluation of the event by the monitor. The overhead is between 28 and 31% for write calls and between 49 and 57% for read calls. This difference results from the structure of the signatures that include an additional assignment or condition in the monitor for the transmitted value at the receiving side (read call).

Table 2 shows the influence of the monitor on the total run-time of the SW-C's runnables. For the ShiftLogic component, two ports have been wrapped and, therefore, the instrumented runnable calls the monitor twice as often as for the other components. The components Transmission and Vehicle, which contain only one instrumented port, therefore, have a smaller run-time overhead.

Memory overhead. We conducted our analysis of the memory overhead using the tool *objdump* of the GNU binutils toolsuite on the compiled object files. Objdump provides a detailed overview of the memory consumption of the text

Table 2: Execution time comparison of original and monitored runnables

	Average			
Component	*Original (ticks)*	*Instr. (ticks)*	*Diff. (ticks)*	*Diff. (%)*
ShiftLogic()	56,03	70,97	14,94	28
Transmission()	57,28	63,77	6,49	11
Vehicle()	48,63	58,39	9,76	20

Table 3: Memory overhead caused by the monitors in byte

	Wrapper				Monitor		
Type	*Read_VS*	*Write_VS*	*Read_G*	*Write_G*	*Contr.*	*ASC*	*IGC*
Code	48	48	48	48	1632	2512	2160
Data	0	0	0	0	88	40	32

(aka. code) section, and the three data sections data, bss (uninitialized data) and rdata (read-only data). The analysis of the memory overhead of the integrated monitors is depicted in Tab. 3. The first four columns show the memory that is consumed by the RTE call wrappers that pass the signals to the monitor. As the functionality of each wrapper is similar, their overhead is constant.

The monitor component consists of a controller (*Contr.*) that manages the monitors and the signatures AllowedSpeedChanges (*ASC*) and InvalidGearChanges (*IGC*). The controller caches the transmitted values, triggers the signature representations, and evaluates their results. The memory overhead of the data section is very small and grows very slowly with the complexity of the signatures because only the state of the monitor and the monitor specific variables, as shown in Sect. 3.1, are stored there.

Scalability analysis. The previous results have shown that the generated monitors can be used in an AUTOSAR environment with reasonable overhead. For the evaluation of the scalability of the monitors, various models of different complexity are generated and the run-time behaviour of the generated C code is measured on a Fujitsu SK-16FX-100PMC evaluation board equipped with an F^2MC-16FX MB96340 series microcontroller (16 bit, 56 MHz). Table 4 shows the different models, the run-time of the generated monitors for processing one event in the signature, where a message consists of a sending and receiving event, and the needed memory on the micro controller. The values for the needed data memory (RAM) include approximately 800 bytes of data that do not origin from the generated monitor. For the measurement of the run-time, 1000 complete runs of the signatures have been performed. Complex conditions and actions in the signature are dismissed and only the monitor itself is measured.

Model 1 to 6 demonstrate how the monitors scale if the number of messages increases in the signature. Model 1 and 7 to 11 show the overhead when all messages have the same message type. In Model 4 and 12 to 14, the number of signatures of constant size (M4) is increased. With an increasing number of processing steps to reach the final state of the signature the initialization

Table 4: Results of the scalability evaluation

Model	*#Messages*	*Different events*	*Run-time/Event in µs*	*Code in byte*	*Data in byte*
M1	1	2	28,384	896	1038
M2	2	4	25,360	1078	1044
M3	3	6	24,277	1230	1044
M4	4	8	23,728	1382	1044
M5	50	100	22,636	8898	1154
M6	100	200	22,660	17026	1268
M7	2	2	25,936	1043	1044
M8	3	2	25,099	1158	1044
M9	4	2	25,176	1562	1044
M10	50	2	46,239	7260	1156
M11	100	2	67,973	13605	1270
M12	2 * M4	8	46,424	2696	1056
M13	3 * M4	8	67,992	4000	1068
M14	4 * M4	8	89,744	5304	1080

overhead gets less important. The code memory consumption increases linearly with the number of messages located in a signature. In all cases, the RAM needed to store the state of the signature increases very slowly, because the state of the signature is binary coded. This is an important factor for use on resource constrained embedded systems. The evaluation shows that the monitors have a constant computing time per processed event (M1 to M6) and only a linear increase for processing an event that is annotated at multiple transitions (M1, M7 to M11).

5 Conclusion and Future Work

In this paper, we have identified and addressed the challenges that emerge from integrating run-time monitoring of complex signatures into the AUTOSAR development process. The presented continuous model-based development process for security and safety monitors (MBSecMon) is integrated into the AUTOSAR process and incorporates data of the AUTOSAR models. As shown, it allows the modeling of monitor signatures in a well comprehensible graphical modeling language and the automatic generation of monitors with a low overhead that fulfill the AUTOSAR conventions. This framework and the generated monitors have been evaluated utilizing an example model provided with the MATLAB suite, for which AUTOSAR code was generated and integrated into an AUTOSAR environment. With the presented approach, we have overcome the lack of support for complex monitoring in the AUTOSAR tool chains. It is applicable to white-box (source code) and black-box (binary) components, as communication between components is intercepted directly at their port interface by instrumentation techniques shown in [17].

In the future, an evaluation in a larger AUTOSAR project with more complex interactions is planned. It has to be evaluated if the MBSecMon process can be used for Logical Program Flow Monitoring [2], eventually with another source specification language such as UML2 activity diagrams or MPNs directly.

Acknowledgements. This work was supported by CASED (www.cased.de).

References

1. AUTOSAR: Specification of Operating System (2011), http://www.autosar.org/download/R4.0/AUTOSAR_SWS_OS.pdf
2. AUTOSAR: Specification of Watchdog Manager (2011), http://www.autosar.org/download/R4.0/AUTOSAR_SWS_WatchdogManager.pdf
3. Cotard, S., Faucou, S., Bechennec, J.L., Queudet, A., Trinquet, Y.: A Data Flow Monitoring Service Based on Runtime Verification for AUTOSAR. In: IEEE 14th International Conference on HPCC-ICESS 2012. pp. 1508–1515 (2012)
4. Cotard, S., Faucou, S., Béchennec, J.: A Dataflow Monitoring Service Based on Runtime Verification for AUTOSAR OS: Implementation and Performances. OSPERT pp. 46–55 (2012)
5. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design 19(1), 45–80 (2001)
6. Frankowiak, M.R., Grosvenor, R.I., Prickett, P.W.: Microcontroller-Based Process Monitoring Using Petri-Nets. EURASIP Journal on Embedded Systems 2009, 3:1–3:12 (2009)
7. Groll, A., Ruland, C.: Secure and Authentic Communication on Existing In-Vehicle Networks. In: Intelligent Vehicles Symposium, IEEE. pp. 1093–1097 (2009)
8. Harel, D., Thiagarajan, P.: Message Sequence Charts. In: Lavagno, L., Martin, G., Selic, B. (eds.) UML for Real, pp. 77–105. Springer (2004)
9. Koscher, K., Czeskis, A., et al.: Experimental Security Analysis of a Modern Automobile. In: IEEE Symposium on SP. pp. 447–462 (2010)
10. Kumar, R., Mercer, E., Bunker, A.: Improving Translation of Live Sequence Charts to Temporal Logic. ENTCS 250(1), 137–152 (2009)
11. Kumar, S.: Classification and Detection of computer Intrusions. Ph.D. thesis, Purdue University (1995)
12. Oh, N., Shirvani, P., McCluskey, E.: Control-flow Checking by Software Signatures. IEEE Transactions on Reliability 51(1), 111–122 (2002)
13. Papadimitratos, P., Buttyan, L., et al.: Secure Vehicular Communication Systems: Design and Architecture. IEEE Communications Magazine 46(11), 100–109 (2008)
14. Patzina, L., Patzina, S., Piper, T., Schürr, A.: Monitor Petri Nets for Security Monitoring. In: Proc. of 1st S&D4RCES. pp. 3:1–3:6. ACM (2010)
15. Patzina, S., Patzina, L., Schürr, A.: Extending LSCs for Behavioral Signature Modeling. In: Camenisch, J., Fischer-Hübner, S., Murayama, Y., Portmann, A., Rieder, C. (eds.) Proceedings of 26th IFIP Sec. pp. 293–304. Springer (2011)
16. Patzina, S., Patzina, L.: A Case Study Based Comparison of ATL and SDM. In: Schürr, A., Varró, D., Varró, G. (eds.) Applications of Graph Transformations with Industrial Relevance. LNCS, vol. 7233, pp. 210–221. Springer (2012)
17. Piper, T., Winter, S., Manns, P., Suri, N.: Instrumenting AUTOSAR for Dependability Assessment: A Guidance Framework. In: 42nd Annual IEEE/IFIP International Conference on DSN. pp. 1–12. IEEE (2012)
18. The MathWorks, Inc.: Modeling an Automatic Transmission Controller. online (2012), http://www.mathworks.de/de/help/simulink/examples/modeling-an-automatic-transmission-controller.html, visited on 12.02.2013