Targeting Freedom from Runtime Errors for AUTOSAR-Based ECU Software

Thomas M. Galla
Elektrobit Automotive GmbH

Alexander Much
Elektrobit Automotive GmbH

ABSTRACT

The automotive standard AUTOSAR provides a standardized basis for electronic control unit (ECU) software development consisting of a layered software architecture consisting of over 80 software modules and libraries accompanied by an associated development methodology.

These software modules and libraries are highly customizable by means of configuration parameters in order to be usable on a broad range of ECUs. Therefore the complexity previously contained in source code is moved to a large degree to the configuration, causing functional requirements of an ECU to be mapped not only to static source code but to configuration parameters as well. This shift of complexity from static source code to configuration leads to a vast set of possible configurations rendering exhaustive testing or verification of the configurations infeasible.

We present an approach for ensuring the freedom from runtime errors for AUTOSAR based ECUs and how to use standardized AUTOSAR XML files to make the verification of software possible and computationally feasible.

Given the fact that more and more automotive ECUs are subject to safety standards, such as ISO/IEC 61508 or ISO 26262, a sound verification strategy that targets freedom from runtime errors is an important part in the safety case.

INTRODUCTION

The automotive standard AUTOSAR provides a broad standardized basis for ECU software development consisting of a layered software architecture with over 80 software modules and libraries accompanied by an associated development methodology. The potential benefits of (re-)using these standardized software modules are undisputed and make the use of AUTOSAR very attractive.

Since the software modules and libraries of the AUTOSAR standard are designed to be usable on a broad range of different ECUs, they are very flexible and to a high degree configurable. Therefore the complexity previously contained in source code is moved to a large degree to the configuration, causing functional requirements of an ECU to be mapped not only to static source code but to configuration parameters as well.

This shift of complexity from static source code to configuration leads to a vast set of possible configurations rendering exhaustive testing or verification of the configurations infeasible. Assuming 80 basic software modules with 50 binary configuration parameters each[1], there are $2^{80*50} \approx 10^{1204}$ possible configuration sets. Given the configuration space, the challenge for the software vendor is to demonstrate freedom from runtime errors for most likely configurations and for the software integrator for each specific ECU configuration.

In this paper we present a stepwise approach for ensuring the freedom from runtime errors for AUTOSAR based ECUs. We point out, how the different verification actions can be (re-)executed in the different phases of ECU development, and illustrate how standardized AUTOSAR XML files can be used to make these verification steps possible and computationally feasible.

This paper is structured as follows: In the first section, the AUTOSAR software architecture and the corresponding development methodology is introduced. The second section focuses on the development life cycle of an ECU based on the AUTOSAR standard. Section three presents a combination of verification methods that need to be applied during the different phases of the development life cycle to target freedom of runtime errors and to construct a sound safety case.

[1] The AUTOSAR software module for signal-based communication alone provides about 180 different configuration parameters.

Section four highlights how the required input for two of these verification methods can be derived from description and configuration files standardized by AUTOSAR. Section five provides a wrap-up and concludes the paper.

AUTOSAR ARCHITECTURE AND DEVELOPMENT METHODOLOGY

AUTOSAR ARCHITECTURE

The AUTOSAR software architecture (see Figure 1) clearly distinguishes between application software components (SWCs) encapsulating all application specific software items (i.e., control loops, interaction with sensor and actuators etc.) and basic software (BSW) modules providing functionality like a real-time operating system (OS), management of non-volatile memory (memory stack), alive supervision with the ECU's watchdog (watchdog stack), communication protocol stacks for the different kinds of automotive communication protocols and networks (communication stack), mode- and state management functionality (mode management stack), and diagnostic services (diagnostic stack). The interface to the hardware is hereby abstracted by the BSW modules of the Microcontroller Abstraction Layer (MCAL) which are the only modules containing hardware dependent code. The runtime environment (RTE) acts as standardized interface between the application software components and the basic software modules, in order to achieve the technical goals of AUTOSAR such as modularity, scalability, transferability and re-usability of application functions developed by different manufacturers and suppliers.

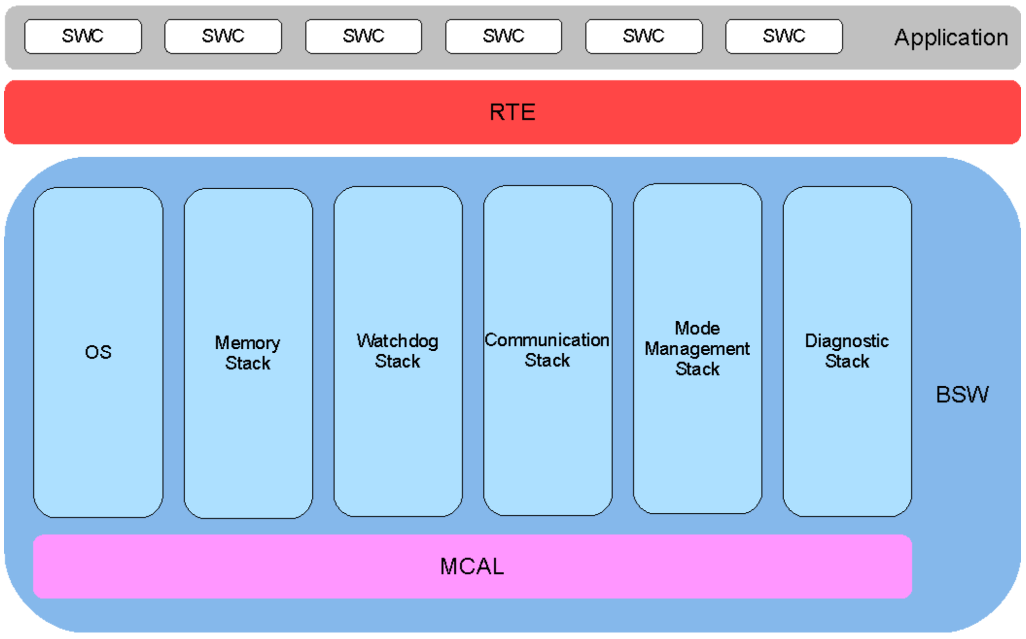

Figure 1: AUTOSAR architecture

The description of interfaces and internal behavior of basic software modules and application software components takes place via standardized XML files, namely basic software module description (BSWMD) [9] and software component description (SWCD) [10]. Amongst other things these files contain information on the provided and required interfaces of a SWC or a BSW module, on the units of execution of SWCs (i.e., so-called runnable entities) and basic BSW modules (i.e., so-called BSW (schedulable) entities), and on the exclusive areas, sequences of code where mutual exclusion must be guaranteed during the execution of the runnable entities and the BSW entities.

DEVELOPMENT METHODOLOGY

In the automotive industry the following two phase development process is commonly accepted. The car manufacturer (OEM) designs the overall electronic system of the whole car, partitions the total functionality into distinct functional units, allocates these functional units to separate ECUs, and defines the interconnection of the ECUs by means of communication networks together with the information exchanged between these ECUs. All this information is usually stored in a design database of the OEM. The OEM then subcontracts the development and the manufacturing of the ECUs to Tier 1 suppliers which develop the ECU autonomously. Once finished, the ECU is then delivered back to the OEM for integration.

AUTOSAR supports this two phase development process by specifying a development methodology (depicted in Figure 2) together with standardized file formats based on XML for the exchange of information between the OEM and the Tier 1.

The system description [8] includes all system-wide parameters, i.e., parameters that are relevant for all ECUs (e.g., the communication matrix and the network topology). The ECU extract of the system description contains only those parts of the system description that are relevant for one specific ECU (e.g., the signals the ECU sends and receives). The ECU extract of the system description is the main piece of information that is exchanged between the OEM and the Tier 1.

The Tier 1 imports this ECU extract into an ECU configuration editor tool, augments this information with ECU local configuration information for the different BSW modules and SWCs producing the ECU configuration. Again the file format for this ECU configuration is based on XML and standardized by AUTOSAR [11].

The ECU configuration is then used as input for configuration code generation tools that produce configuration data structures for the static BSW module and SWC code. Additionally this generation tools create the SWCD and the BSWMD depending on the ECU configuration. The file format for both SWCD and BSWMD is XML-based and standardize by AUTOSAR as well. The RTE generator uses the SWCDs and BSWMDs together with the ECU configuration as input to produce the RTE code.

In a last step, the RTE code, the static BSW module and SWC code, and the configuration data structures are compiled and linked together to create the final executable.

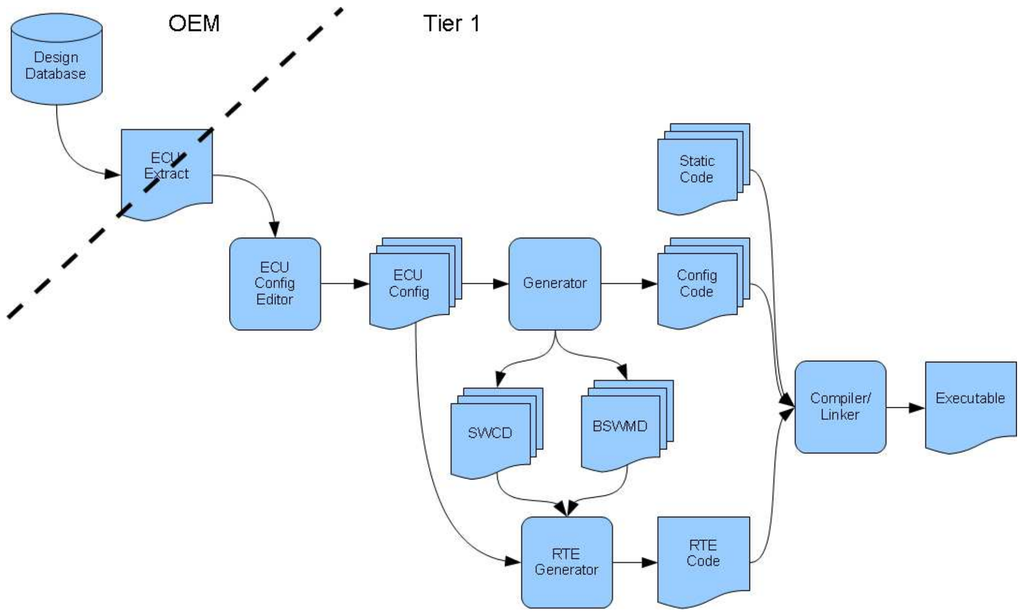

Figure 2: AUTOSAR Development Methodology

The static BSW module code is hereby usually provided by a dedicated software vendor together with the ECU configuration editor, the configuration generation tools and the RTE generator. A very common approach for developing the static SWC code is the use model-based design techniques together with automatic code generation.

AUTOSAR ECU DEVELOPMENT LIFE-CYCLE

The development life-cycle of an AUTOSAR-based ECU can be characterized by the reduction of the amount of variants until a specific set of variations and a specific configuration is chosen for the concrete ECU project. This also determines the relationship of the involved parties in an ECU project and in turn the verification responsibilities (see Figure 3). The version of the AUTOSAR standard is chosen during module development, the set of BSW modules, the target hardware and the compiler are chosen during integration and the configuration for a specific ECU is determined during ECU development.

During BSW module development the BSW modules need to be verified against a large amount of variants. Verification activities are conducted mostly “context free”, i.e. the same software is used for a rear-mirror or a highly critical ECU in the powertrain. As development progresses the development activities become more ECU specific. In turn the verification of ECU functionality needs to be performed iteratively by the software integrator as more functionality is added to the ECU configuration for each milestone. This means that some of the verification methods discussed below need to be re-performed by the software integrator using the specific ECU configuration for each development milestone. With each iteration the verification activities shift from being “context-free” to being more and more “context-aware” and verifying the functionality of the specific ECU.

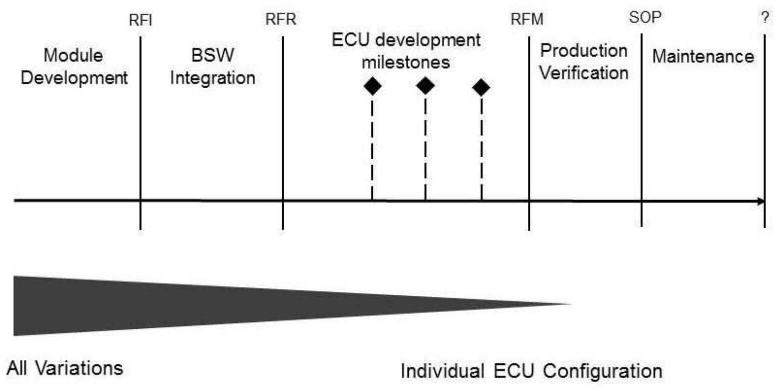

Figure 3: ECU Development Life-Cycle

VERIFICATION METHODS

CHOOSING AND COMBINING VERIFICATION METHODS

Complex software architectures like AUTOSAR are complicated to validate and verify. But assuring a high degree of confidence in the correct functionality as well in freedom from runtime errors is essential for an implementation that is going to be used in possibly safety-relevant ECUs with very large production numbers.

Such systems usually undergo different phases of validation and verification performed by all parties involved in producing them. Choosing the optimal verification measures, combining them and performing them at the right phase during the development is therefore crucial for achieving the required confidence level as well as keeping in control of costs of such activities. Also the distribution of verification activities between all parties involved is important to avoid unnecessary measures that are simple duplication and don’t lead to a better error detection rate.

RESTRICTING THE CONFIGURATION SPACE

The large amount of configuration variants in AUTOSAR projects contain a large number of inconsistent or “nonsense” variants that may lead to issues in the BSW modules. To limit these inconsistent variants the EB configuration tooling for AUTOSAR BSW supports the software integrator with an automated configuration of a majority of the relevant parameters e.g., by importing network data contained in the ECU extract of the system description [8] or in a FIBEX [7] file to properly configure the ECU’s communication protocol stacks.

The remaining part of the configuration parameters (i.e., those that cannot be filled automatically) contains a subset of parameters whose value can be derived via some formula from the values of other (already configured) parameters. – Here the EB configuration tooling provides means for automatic calculation of these parameters. Another part of the parameters that cannot be filled automatically can however be set to OEM specific default values. For this purpose the EB configuration tooling provides the possibility to define pre- and recommended configurations from which the software integrator can choose.

The resulting reduction in manual configuration lowers the possibility of inconsistent or possibly dangerous configuration variants. During the configuration process and prior to the generation of the C-code, the configuration itself is verified against a set of rules to lower possible inconsistencies produced during manual configuration that may lead to incorrectly generated code or possibly to dangerous code.

USE-CASE BASED TESTS AND CONFIGURATIONS

Use-cases are the basis for creating ECU configurations for validation and verification purposes. To keep verification efficient, the amount of test configurations should be as small as possible, but large enough to cover the configuration space. The coverage of the configuration space can be measured using an approach that is similar to structural code coverage measurement for source code by measuring the coverage of the code generator. Having a sufficiently large set of test configurations is the basis for the methods below as e.g., static or dynamic analysis needs to be performed for each individual test configuration.

STATIC ANALYSIS

Static analysis has become an invaluable method in software development and tool support for static analysis keeps getting better. Static analysis can be used for many purposes, but many run-time errors can already be detected e.g. when checking compliance with coding guidelines like MISRA-C:2004 [2]. Static analysis is performed without executing the software and can identify errors, but often not all of them, e.g. uninitialized variables can usually only be detected if a local variable is not analyzed.

Because many configuration options of BSW modules lead to different pre-compile time variants, e.g. by using the pre-processor, the static analysis needs to be performed for all use-case configurations and then accumulated to one complete analysis report. This also means that for the final configuration the software integrator needs to perform static analysis again, because new violations could have been introduced by the ECU configuration.

Static analysis is primarily a method for error prevention by checking for potentially unsafe code constructs that can be used very early in the development process: in the end it is a bad development strategy to introduce such errors in the first place and passing them to software test teams to find them. If used early during development, the amount of effort spent for other verification methods is significantly lowered, leading to a more efficient and effective verification process. Modern safety standards mandate the use of static analysis and a large part of methods required by such standards for software development can be covered with static analysis.

DYNAMIC ANALYSIS

Due to the abstraction provided by the BSW modules of the MCAL most software in AUTOSAR-based ECUs is actually independent from the target hardware. Thus dynamic analysis can be conducted on a “virtual ECU”, e.g. using a common desktop system. This enables the use of verification techniques that are not available on the target hardware, e.g. bounds-checking tools that instrument the source code and/or the object code. Bounds-checking tools [3][4][5][6] insert checking code for e.g. assignments to pointers, dereferencing of pointers, array accesses, stack protection, etc.

When executing test cases the source code as well as the test code is checked for any run-time errors during execution. This is an important aspect, since there often is more test code than code and errors in test code can lead to a failure in detecting errors in the source code itself. Safety standards require the measurement of the structural code coverage of software tests and the structural coverage measurement can then be used as completeness argument for dynamic analysis, e.g. by bounds-checking tools.

DEFENSIVE PROGRAMMING

Defensive programming is a technique that performs additional range and consistency checks during run-time. AUTOSAR supports defensive programming via the Development Error Tracer (DET) module and by specifying the required checks for each BSW module. The main focus of the AUTOSAR DET module is the detection of integration and configuration errors during development and integration. Once the ECU software has been properly integrated and tested, these additional checks are usually deactivated in production code to reduce the overall execution time.

EB has extended the concept by checking additional preconditions, invariants, unreachable code parts and post-conditions. This brings the verification more in-line with semi-formal verification methods like “design by contract” et al. [1]. Using defensive programming methods the BSW modules achieve a high level of robustness and error detection capability. In safety relevant ECUs the “critical” SW components and code paths often include the online checks also for production code, which can serve as part of an argument in the safety analysis.

SEMANTIC ANALYSIS

Semantic analysis is performed by using abstract interpretation [12], which aims to provide a sound approximation of the semantics of a program and is the only viable method that ensures that 100% coverage of all paths through a program can be guaranteed. Semantic analysis can be computationally problematic and time-consuming if used for a large code base and for code that didn’t pass other verification methods mentioned above. The strategy should always be to perform a static analysis before the semantic analysis to reduce computational and manual analysis effort.

Also defensive programming is very supportive for semantic analysis as e.g. the ranges of variables are limited, which helps to reduce the configuration effort for tooling and the amounts of false positives significantly. Semantic analysis is performed using few “full” configuration sets for each BSW module and to support the verification of robustness of BSW modules and all external interfaces of a BSW module are called “full-range” and without initialized global variables.

In order to verify the specific ECU software, the analysis itself needs to be performed again by the software integrator using the final configuration of the ECU project. Using AUTOSAR BSWMDs as basis, parts of the configuration for semantic analysis tools can be automatically generated (see below) to keep the ECU configuration and the tool configuration consistent.

SOFTWARE TESTING

Software testing is of course an important part of the verification and identifies run-time errors. Exhaustive testing of a complete AUTOSAR-based ECU is infeasible as the amount of possible configuration variations is too large. Thus software tests in the context of this paper are mainly used as basis for other verification techniques. However, many verification methods rely on an argument for completeness of the software tests at the different levels in the V-model and the most common techniques and metrics apply: organized derivation of test cases from requirements and measurement of structural code coverage.

The experience has shown that using the verification methods detailed above, testing becomes more effective and especially the cost of more expensive testing at a higher level in the V-model is reduced. This effect is especially important for the software integrator: the depth of testing is not reduced, but the software integrator can focus on the verification of the specific ECU project and expensive cycles of re-integrating new BSW modules are reduced.

CONCURRENCY ANALYSIS AND OPTIMIZATION

The AUTOSAR standard does not specify in detail the scheduling of the runnable entities of SWCs or the schedulable BSW entities of the BSW modules: the scheduling is configurable by mapping both kinds of entities to one or more operating system tasks. Concurrency protection is provided by configurable exclusive areas that can be mapped to operating system mutual exclusion primitives (like semaphores) or interrupt locking mechanisms or even completely omitted in case mutual exclusion is guaranteed by design (e.g., if mutual preemptions can be ruled out because of the appropriate selection of task priorities). Selecting the right kind of mutual exclusion primitives for the implementation of an exclusive area is a complex issue for the software integrator as a safe and optimal configuration requires detailed knowledge about the internals of BSW modules. Basically there are three problems:

1. unbalanced use of exclusive areas,
2. too few exclusive areas or
3. too many exclusive areas in the specific ECU configuration.

Unbalanced exclusive areas can lead to permanently locked interrupts or sometimes even to CPU exceptions, if a double release of interrupt locks is considered an exception by the CPU. The obvious methods against having unbalanced exclusive areas are coding guidelines and code inspections against them. Another method is the stubbing of mutual exclusion primitives with supervising code that monitors that the primitives are used in a balanced fashion and then executing all use-case tests using the structural code coverage of these tests as completeness argument.

Missing exclusive areas can lead to inconsistent or corrupt data and in turn to unpredictable behavior of the ECU. Such issues cannot be efficiently and fully detected using software testing as the test case is required to produce the error using a scenario of parallel execution of a very specific code line, sometimes even a specific assembler instruction. As a consequence these problems are often sporadic, hard to analyze and reconstructing the conditions leading to the problem is usually expensive. Missing exclusive areas can be identified using abstract interpretation and an activation scenario that resembles the specific ECU configuration as closely as possible.

Having too many exclusive areas can lead to a non-performing system as the execution of them can be time consuming. The performance degradation may be tolerable in systems where fast interrupt locks are available, which is usually the case for non-safety ECUs, but safety-relevant ECUs often need to apply independence to achieve the desired (A)SIL level. Such systems use partitioning (see Figure 4) to separate safety from non-safety code, e.g. to perform (A)SIL decomposition in software or simply to enable effective safety analyses by limiting the amount of code to be analyzed.

On such systems non-safety code is not allowed to access hardware mechanisms such as fast interrupt locks as this would allow non-safety code to disable interrupts and thus prevent safety-related code and/or the operating system from being executed. Therefore the mutual exclusion primitives that implement an exclusive area are required to use operating system mechanisms in order to cross memory protection boundaries, which could result in two rewrites to the memory protection unit. Experience from projects has shown that for such systems optimization of exclusive areas is a key issue to meet the timing constraints of the ECU.

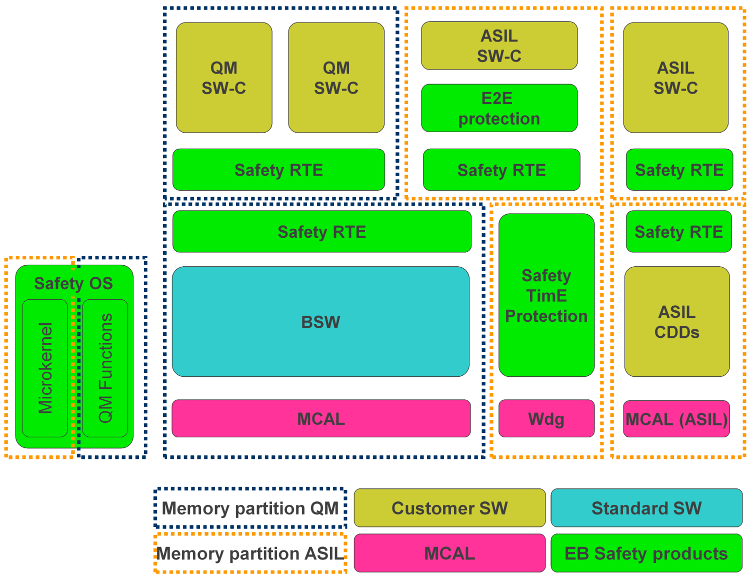

Figure 4: Safety architecture of an AUTOSAR-based ECU using memory and execution protection

Furthermore for safety-relevant ECUs predictability is a major issue. Using a static schedule with minimal jitter is the key to predictability. Each exclusive area however potentially increases the worst case execution time of a task by the worst case blocking time cause by another task that shares the same exclusive area thus resulting in considerable execution time jitter. This way, small changes in the execution time of code can sum up to significant differences reducing the predictability of the overall ECU software. Optimizing exclusive areas thus simplifies worst case execution time analysis, leads to a more predictable scheduling in the ECU and makes it possible to reduce “safety” margins in scheduling and in turn optimizes CPU time usage.

INITIALIZATION AND CONCURRENCY ANALYSIS SUPPORT IN AUTOSAR 4.X

As previously stated the semantic analysis and concurrency analysis can be computationally problematic and time-consuming especially when applied to a software modules and libraries that are highly customizable as it is the case with AUTOSAR. Thus it is imperative to reduce the problem space for both techniques by applying the following two strategies:

1. limiting the ranges that values of variables can assume by explicitly defining these (limited) ranges and/or by specifying proper initialization values for the variables
2. limiting the potential concurrent threads of execution

The input for both strategies can be retrieved from the XML files standardized by AUTOSAR, namely the ECU configuration, the BSWMD and the SWCD.

USING CONFIGURATION INFORMATION, BSWMD, AND SWCD

The input needed for the first part of strategy 1 can be obtained from the source code itself (in case the respective variables are of enumeration type) of from the SWCD in case the variables are from integral type, but only a limited range of values is allowed. The same is true for the input of the second part of strategy 1: In case variables are explicitly initialized in the source code, the initialization value is simply contained in the source code. If this is not the case, the BSWMD needs to be consulted to decide whether the variable is mapped to a memory section that is initialized with zero as part of the ECU’s startup code (in this case the initialization value of the variable is zero) or whether the variable is mapped to a memory section that is uninitialized (in that case an analysis tool has to make a worst case assumption based on the type of the variable) to be able to derive a correct initialization scenario.

The input needed for strategy 2 needs to be collected from the BSWMD and the RTE’s and the OS’s ECU configuration (in case of BSW modules) and from the SWCD and the RTE’s and the OS’s ECU configuration (in case of SWCs). The ECU configuration of the OS contains information on the used scheduling tables (in case of time-driven scheduling), the chosen scheduling policy (preemptive vs. non preemptive) and the selected task priorities. The ECU configuration of the RTE contains information on the mapping of runnable entities and BSW entities to OS tasks. The BSWMDs and the SWCDs contain information on the call graphs of the different runnable entities and BSWM entities together with the information in the exclusive areas accessed by these entities. Based on this information a restricted activation scenario can be derived limiting the potential concurrent threads of execution to only those combinations that are actually possible in the given configuration.

KEEPING AUTOSAR AND ANALYSIS TOOL CONFIGURATIONS CONSISTENT

One of the key issues in the process described above is to ensure consistency between the ECU configuration, the contents of the BSWMD and the SWCD, and the derived initialization and activation scenarios provided as input for the tool conduction the semantic and concurrency analysis. Having a manual step here in the development and verification process is tedious and error prone.

The transformation of the relevant information contained in the ECU configuration, the BSMD, and the SWCD into initialization scenarios and activation scenarios is conducted in an automated fashion in parallel to the generation of RTE code and configuration code making it easy to re-run the semantic and concurrency analysis after each relevant change in the ECU configuration.

ENSURING SAFETY OF EXCLUSIVE AREA OPTIMIZATIONS

To reduce the overall execution time the RTE can remove mutual exclusion primitives for implementation of exclusive areas if mutual exclusion is guaranteed by some other means (e.g., by proper time-driven scheduling or by adequate choice of priorities to prevent mutual preemption) thus providing a safe optimization. The information required to be able to judge whether such an elision is a safe optimization is again contained the ECU configuration of the OS, the ECU configuration of the RTE, the BSWMDs, and the SWCDs.

In order to construct a sound line of argument that these optimization performed by the RTE really are safe, the following needs to be ensured:

- The information contained in the BSWMDs and the SWCDs, namely the call graphs and the access to exclusive areas must match the real behavior of each BSW module and each SWC.
- The OS must activate the OS tasks correctly according to the respective configuration
- The algorithm for elision of the mutual exclusion primitives must be implemented correctly
- The generated RTE must adhere to the configured mapping of runnable entities and/or BSW entities to task

The first item can be ensured by instrumenting the entry and exit points of the functions of each BSW module and the each SWC as well as the mutual exclusion primitives in a way that the call to these functions/primitives can be logged. – Using this instrumented code in execution of the tests derived from the previously described use-cases thus generates all possible call traces of the functions and the mutual exclusion primitives. Comparing these call traces against the information contained in the BSWMDs and the SWCDs ensures that the BSWMDs and SWCDs are correct and complete w.r.t. call graphs and the accesses to exclusive areas.

The second item is usually ensured by using an OS that has been developed to meet the desired safety integrity level (e.g., the microkernel OS depicted in Figure 4) and is thus considered to be free from design and implementation faults.

The third and the fourth item can be ensured, by re-applying the semantic and concurrency analysis previously described to the optimized RTE code (where mutual exclusion primitives possibly have been elided) and comparing the analysis result to the analysis result of the non-optimized code. In case only safe optimizations have been performed, both results must be equal.

CONCLUSION

Achieving perfect freedom from runtime errors in a highly configurable and complex architecture such as AUTOSAR is an important goal, because the implementation is going to be used in many different ECUs and application domains. Many of these ECUs are produced on a large-scale leading to the situation where the likelihood of a runtime error being triggered and leading to a negative effect on the environment is large.

With the widespread adoption of functional safety standards in the automotive domain, the verification methods need to be adapted, and many ECU development projects require an argument for demonstrating freedom from runtime errors, such as the one presented in this paper.

Traditional verification methods such as software testing, but also formal methods do not efficiently scale to the amount of configurations options and complexity found in an AUTOSAR architecture. Verification tools and methods have become more sophisticated and effective in the last decade and a well-thought combination of methods and use of tools can now result in a high degree of confidence for being “nearly” free from runtime errors.

ACKNOWLEDGMENTS

The authors want to thank the ECU development teams at EB for their great ideas and their contributions over many years for a continuously improving AUTOSAR implementation.

REFERENCES

[1] Bertrand Meyer, *Applying "Design by Contract"*, in *Computer (IEEE)*, vol. 25, no. 10, October 1992, pages 40-51.
[2] MIRA Ltd., *MISRA-C:2004 - Guidelines for the use of the C language in critical systems*, ISBN 0952415623, October 2004
[3] Frank Ch. Eigler, *Mudflap: Pointer Use Checking for C/C++*. In Proceedings of the GCC Developers Summit 2003, pages 57-67,May 25–27, 2003, Ottawa, Ontario, Canada
[4] Nicholas Nethercote and Julian Seward. *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*. In Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007.
[5] Julian Seward and Nicholas Nethercote, *Using Valgrind to detect undefined value errors with bit-precision*. In Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, April 2005.
[6] Richard W. M. Jones and Paul H. J. Kelly. *Backwards-compatible bounds checking for arrays and pointers in C programs*. In M. Kamkar and D. Byers, editors, Third International Workshop on Automated Debugging. Linkoping University Electronic Press, 1997
[7] ASAM FIBEX Expert Group, *ASAM MCD-2 NET - Data Model for ECU Network Systems (Field Bus Data Exchange Format)*, Version 4.0.0, September 2011
[8] AUTOSAR, *System Template (Document Version 4.2.0)*. Release: 4.0 Revision: 3.
[9] AUTOSAR, *Specification of BSW Module Description Template (Document Version 2.2.0)*. Release: 4.0 Revision: 3.
[10] AUTOSAR, *Software Component Template (Document Version 4.2.0)*. Release: 4.0 Revision: 3.
[11] AUTOSAR, *ECU Configuration Parameter Definition (Document Version 4.2.0)*, Release: 4.0 Revision: 3.
[12] Patrick Cousot and Radhia Cousot, *A gentle introduction to formal verification of computer systems by abstract interpretation*, Marktoberdorf summer school, 2009.

CONTACT

Thomas M. Galla (thomas.galla@elektrobit.com)

Thomas M. Galla studied computer science at the Technical University of Vienna. He received the M.Sc. degree (with distinction) in 1995, started working as a research assistant in the area of distributed real-time systems in 1996, and finished his Ph.D. (with distinction) in 2000. In September 2000 Thomas M. Galla joined DECOMSYS, now Elektrobit Austria, where he is responsible for the software architecture of EB's AUTOSAR software stack and EB's AUTOSAR tooling as a senior software architect.

Alexander Much (alexander.much@elektrobit.com)

Alexander Much studied mathematics at the University of Erlangen-Nuremberg and Cambridge (UK). He received a Dipl.-Math and M.Sc degree. In 2000 he joined SuSE Linux and in 2004 Elektrobit Automotive GmbH (formerly 3SOFT GmbH), where he is responsible for the quality management of the development for automotive ECUs.

ADDITIONAL SOURCES

Alexander Much, *EB tresos AutoCore and Polyspace - Measuring AUTOSAR software with a view towards ISO 26262*, Polyspace user meeting 2011, http://www.mathworks.de/company/events/conferences/polyspace-user-meeting/

Alexander Much, *ISO-26262-konforme Komponenten ohne Laufzeitfehler*, Elektronik Praxis 2011/12, http://www.elektronikpraxis.vogel.de/embedded-computing/articles/342436/