Formal Methods in Computer-Aided Design 2021

Model Checking AUTOSAR Components with CBMC

Timothee Durand[*], Katalin Fazekas[†], Georg Weissenbacher[†] and Jakob Zwirchmayr[*]
[*]TTTech Auto AG, Vienna, Austria
[†]TU Wien, Vienna, Austria

***Abstract*—Automotive software needs to comply with stringent functional safety standards to reduce the risk of malfunction. In particular, the ISO 26262 standard highly recommends the use of formal verification for highly safety-critical software components. Automated formal verification techniques (such as Model Checking) enable the quick detection of intricate software bugs and can, to a limited extent, even guarantee their absence.**

We report our efforts to deploy the openly available verification tool CBMC to verify AUTOSAR Software Components and Complex Device Drivers using Bounded Model Checking and k-induction combined with upfront static analysis.

I. Introduction

Modern cars now contain as many as 150 Electronic Control Units (ECUs) running software from different suppliers. AUTOSAR, an open and standardized software architecture for automotive applications, guarantees the interoperability of automotive software components. This platform provides a common development methodology based on a standardized exchange format for describing software components (ARXML), standardized communication interfaces and a Run-Time Environment (RTE), and a basic software (BSW) layer (see Fig. 1). The BSW comprises hardware-specific software modules (including Complex Device Drivers (CDDs)) that provide functions to the upper software layers. The RTE middleware provides interfaces and functions for inter- and intra-ECU communication between the application software components. Software Components (SWCs) in the application layer access the lower layers via the RTE, and can hence be readily deployed on different vehicle and platform variants.

The ISO 26262 [1] functional safety standard establishes safety requirements for automotive components (including software). The norm defines four Automotive Safety Integrity Levels (ASILs) ranging from A (low risk) to D (life-threatening hazards). ASIL-D requires the highest degree of rigor, including (semi-)formal verification in the development process. Consequently, formal methods are frequently applied in industrial dependable system design [2]. Moreover, ASIL-code needs to be reverified whenever the implementation is changed, re-generated, or re-configured.

In this context, automated static analysis techniques (such as abstract interpretation or software model checking [3], [4]) are particularly attractive, as they require comparatively little manual interaction and can detect intricate software bugs and, to a limited extent, even guarantee their absence.

We investigate the applicability of model checking to AUTOSAR code written in ANSI-C. While commercial tools for static analysis of AUTOSAR code exist [5], we focus on the software model checking tool CBMC [6] because of the tool's availability, sustained development, and its permissive open source license. The latter allowed us to adapt CBMC to our work-flow and requirements: the specifics of AUTOSAR software and the ISO 26262 requirements (such as the ARXML description, the use of the RTE, and repeated verification runs) imposes the need for an automated tool chain.

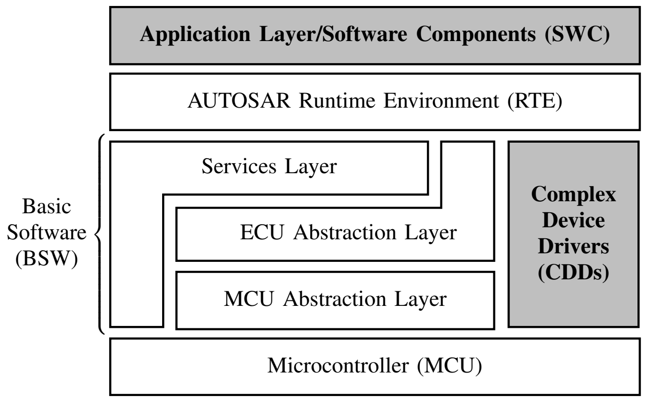
Application Layer/Software Components (SWC)
AUTOSAR Runtime Environment (RTE)
Basic Software (BSW)
Services Layer
ECU Abstraction Layer
MCU Abstraction Layer
Complex Device Drivers (CDDs)
Microcontroller (MCU)

Fig. 1. AUTOSAR Architecture

Contributions. Our report (based on the master's thesis of the first author [7]) describes the following contributions:

1) To apply CBMC to AUTOSAR code, we generate a test harness and RTE-stubs from an ARXML description.
2) We deploy Bounded Model Checking (BMC) to detect bugs, k-Induction to prove their absence, and combine both techniques with an upfront static analysis to improve verification performance and results.
3) We present case studies for SWCs and CDDs and discuss the different challenges regarding their verification.
4) We report our learned lessons and the practicality of the approach and identify open challenges and future work.

II. Methodology

To verify our SWCs and CDDs (described in subsect. III-A), we need to (1) generate the verification environment and (2) instrument and augment the code with static analysis results.

A. The AUTOSAR Platform

AUTOSAR uses three abstraction levels to describe the SWCs of a system. The highest level—the Virtual Function

https://doi.org/

```
int main_k_base() {
  SWC_Init();

  for(i=0; i < K; i++) {

    SWC_Step();
    assert(P);
  }
}
```

```
int main_k_step() {
  SWC_Init();
  mod_ndet_loop_variables();
  for(i=0; i < K+1; i++) {
    assume(P);
    SWC_Step();

  }
  assert(P); }
```

Fig. 2. Entry points for k-Induction experiments to prove property P

Bus (VFB)—describes types of SWCs and their connections to other SWCs (PortInterfaces and PortPrototypes), as well as the messages they exchange via their ports (DataTypes). At the middle level—the RTE—the execution behavior of SWCs, i.e., RunnableEntities and their trigger events, are defined. Finally, at the implementation level, these defined RunnableEntities are mapped to their implementations (given as source or object code).

System constraints and the system configuration are described in the ARXML format (see Fig. 3 for an example). In the given context, the SWC Description and the RTE Extract of the ECU Configuration are of relevance, since they describe the messages and data-types that SWCs can exchange.

B. Generating Verification Environment

The RunnableEntities of an SWC (defined in the corresponding ARXML model [8]) provide initialization and step functions, which are invoked periodically in an order we presume to be fixed (see also sect. V).

BMC focuses on checking the correctness of the program only up to a predetermined number of iterations of each loop, pruning all executions that require more. The entry point of our generated test harness for BMC is a function which, after initialization, calls the step functions of the RunnableEntities in an (unbounded) loop.

The test harness for k-Induction[1] has two entry points: one for the base case and another for the inductive step. Fig. 2 illustrates the principle of k-Induction: BMC is used to establish the base case by checking whether the assertion P holds for the first K loop iterations. Subsequently, we use BMC to check whether P holds after $K + 1$ steps under the assumption that it holds in the first K iterations starting from an *arbitrary* program state. If both the base case and induction step succeed, then P holds after any number of loop iterations.

SWCs exclusively interact with each other and with the BSW through the RTE (see Fig. 1), and RTE ports are their only external input [9]. We assume the correctness of the RTE implementation and replace it with an appropriate abstraction. This has two consequences: Firstly, it results in a smaller code base that is more tractable for verification tools. Secondly, as our RTE abstraction conservatively models the most general environment of the SWC, it takes arbitrary interactions with the environment (e.g., any communication via the RTE) into account. This modular approach guarantees that a change in

[1]CBMC's built-in support for k-Induction did not cope with the nested loops in our SWCs, which is why we require a separate harness.

```
 1 <IMPLEMENTATION-DATA-TYPE UUID="...">
 2   <SHORT-NAME>Dt_Engine_RPM</SHORT-NAME>
 3   ...
 4   <COMPU-METHOD-REF DEST="COMPU-METHOD">
 5      /DataTypes/CompuMethods/CM_Engine_RPM
 6   </COMPU-METHOD-REF>
 7   <IMPLEMENTATION-DATA-TYPE-REF DEST="...">
 8      /AUTOSAR_Platform/ImplementationDataTypes/uint16
 9   </IMPLEMENTATION-DATA-TYPE-REF>
10   ...
11 </IMPLEMENTATION-DATA-TYPE>
12 ...
13 <COMPU-METHOD UUID="...">
14   <SHORT-NAME>CM_Engine_RPM</SHORT-NAME>
15   ...
16   <COMPU-SCALE>
17     <LOWER-LIMIT INTERVAL-TYPE="CLOSED">0</LOWER-LIMIT>
18     <UPPER-LIMIT INTERVAL-TYPE="CLOSED">255
19          </UPPER-LIMIT>
20     <COMPU-RATIONAL-COEFFS>...</COMPU-RATIONAL-COEFFS>
21   </COMPU-SCALE>
22   ...
23 </COMPU-METHOD>
```

```
   i void modif_nondet_Dt_Engine_RPM(Dt_Engine_RPM* tmp);
  ii void modif_nondet_uint16(uint16* tmp);
 iii Std_RetType get_nondet_Std_ReturnType();
  iv Std_RetType
   v   Rte_Read_Engine_RPM_stub(Dt_Engine_RPM* tmp);
  vi
 vii void modif_nondet_Dt_Engine_RPM(Dt_Engine_RPM* tmp) {
viii   modif_nondet_uint16(tmp);
  ix   assume(0 <= *tmp && *tmp <= 255);
   x }
  xi
 xii Std_RetType
xiii Rte_Read_Engine_RPM_stub(Dt_Engine_RPM* tmp){
 xiv   modif_nondet_Dt_Engine_RPM(tmp);
  xv   return get_nondet_Std_ReturnType();
 xvi }
```

Fig. 3. Parts of ARXML specification of data type Dt_Engine_RPM (above) and an example of using it in generated RTE function stubs (below)

the environment (e.g., the deployment of other components) does not invalidate prior verification results.

The ARXML specification [10] and the AUTOSAR meta model [8] describe the DataTypes of messages, allowing us to automatically generate an abstraction of the RTE communication functions. Fig. 3 depicts parts of a specification in the ARXML format that defines data types on different abstraction levels. Lines 7-9 state that Dt_Engine_RPM is implemented as uint16. Lines 4-6 refer to a CompuMethod element that specifies a range of valid values from 0 to 255 for the data type. These limits guarantee that the computation will result in a value representable by uint16. For a thorough definition of data types and their constraints see [8, Sect. 5].

In our RTE abstraction parameters and return values of RTE functions are first havoced and then constrained based on information provided in the ARXML specification. These constraints are automatically generated. We generate non-deterministic modifier and generator functions that are invoked in the generated RTE API stubs (see, e.g., function Rte_Read_Engine_RPM_stub in Fig. 3). Fig. 3 also illustrates how the data constraints defined by the XML in lines 17-18 translate into a C assumption (line viii) due to the type Dt_Engine_RPM.

C. Static Analysis and Instrumentation of Code

As a next step, the verification target SWC source code, its dependencies and the generated RTE stubs are built and linked into a single object with CBMC. Though our software project is complex and uses many architectural parameters, CBMC's `goto-cc` could seamlessly replace the compiler and linker in our build process. We note that, in accordance with the ISO 26262 standard, our code base is written in a well-specified and supported sub-set of the ANSI-C language.

Before starting the verification with CBMC, we perform an upfront static analysis of the code to support and complement the strengths of CBMC. To this end, we emit the complete target project into a single source file and run Frama-C [11] on the resulting code. While Frama-C provides a wide range of static analysis techniques, we only employed its Evolved Value Analysis (EVA [12]) plug-in, which is based on abstract interpretation techniques. We used its default parameters that do not rely on more advanced abstract domains. This analysis can infer relatively small value sets for the variables (including function pointers), which simplifies the task of CBMC, but also provides indispensable type constraints for constructing induction proofs in some of our k-Induction experiments. The results of the static analysis are automatically incorporated as assumptions constraining the values of global variables (which represent the entire state of the system) and as replacements of function pointers with explicit case statements.

Prior to instrumentation of the code with the constraints provided by Frama-C, we verify (in independent k-Induction runs) that the value sets provided by Frama-C are actually inductive invariants. To verify the results of the function pointer analysis, the bodies of functions that are unreachable according to Frama-C are replaced with failing assertions which are then checked using CBMC.

D. Implementation details

To automatically parse the ARXML specifications, RTE headers and to generate C stubs, we relied on several openly available Python modules (e.g. PyCParser [13], lxml [14], and cogu-autosar [15]). Some missing POSIX stubs were implemented manually, and we had to patch CBMC to emit proper C code for the SWCs in our experiments.

III. Case Studies

A. Component Descriptions

We analyse four AUTOSAR SWCs of an automotive software platform that comprises of ECUs with multiple hosts. The platform provides services such as a common time-base for the hosts, global time-triggered scheduling, and time-triggered or time-sensitive communication between hosts. A custom RTE hides the fact that the underlying system is distributed and hosted on multiple SoCs/CPUs from the Application SWCs.

LifeCycle Service Server (LCS-S) component: This component is typically executed on the host with the highest ASIL and implements a state machine that determines the state (`Init`, `Standby`, `Running`, etc.) of each host. `Running`, for instance, indicates that the platform started up successfully and all hosts are operating under supervision. State transitions are triggered by failing built-in self tests, or depend on the states of other services. The LCS-S sends requests to its clients to trigger transitions and ensures that all client hosts transition correctly and report the expected lifecycle states.

While the LCS-S communicates with other SWCs via the RTE, it is considered a CDD because it directly interacts with other health- and safety-related platform services implemented as CDDs. These interactions via non-standardized interfaces require a few LCS-specific extensions of the verification environment and hence knowledge about implementation details.

LifeCycle Service Client (LCS-C) component: implements the same state machine as the LCS-S and periodically checks whether state transitions are required or have been requested by the LCS-S. An example for a transition requested by the LCS-S and confirmed by the LCS-C is the power-off sequence, where clients might store data in non-volatile memory.

Vehicle Communication Service (ApCom) component: This Application SWC is typically either ASIL-B or D and receives messages from the CAN bus (via the corresponding service in the BSW) and transforms them into RTE data types. Thus, the developers need not be aware of the underlying CAN specifics.

As ApCom utilizes only RTE and BSW COM interfaces, it can be model checked with a generic abstraction of these interfaces. Since large parts of the configuration and the implementation are generated based on a mapping between the CAN and RTE messages, the repeated (automated) verification of this generated code is frequently necessary.

Middleware: This component is a CDD that communicates with other hosts through a Transport Layer (e.g. Ethernet or a time-sensitive version thereof), often relying on OS system calls. Since the exchanged messages contain RTE data, it requires non-standardized interaction with the RTE (such as access to its buffer management system), which complicates verification. While the implementation of the buffer management is static, generated or configurable parts of the code introduce the need for repeated analysis. Since it handles ASIL data, the Middleware may be classified up to ASIL-D.

Table I presents some code metrics for each SWC to illustrate their complexity. More details are available in [7, Section 5]. The components of the LifeCycle service are simpler than the other SWCs, with the LCS-S being the more complex one of both due to supervision and platform initialization tasks. The ApCom component relies heavily on calls-by-reference and function pointers, as evidenced by the amount of pointer arithmetic and dereference operations. Its buffer and data frame manipulation operations make the Middleware the most challenging component of our case study. The high complexity metrics for ApCom and Middleware also denote the presence of large chunks of generated code with repetitive structures within these components.

B. Checked program properties

Our goal is to automatically detect potential errors and vulnerabilities (expressed as assertions) in our code base. In addition to assertions added by developers, we check the

TABLE I
CODE METRICS OF TARGET SOFTWARE COMPONENTS

		LCS-C	**LCS-S**	**ApCom**	**MW.**
Operations	Pointer dereference	50	115	2222	2170
	Add. & Subst.	31	129	330	3662
	Mult. & Div.	36	76	898	471
	Bitwise operations	10	14	11	304
Control flow	If statements	119	243	1276	948
	Loops	4	17	77	76
	Function calls	129	309	1347	1328
	Function returns	66	136	365	329
Complexity	Lines of code	1469	4923	15973	16536
	Program locations	529	1182	5935	7061
	Global variables	34	94	427	584
	MacCabe Cycl. Compl.	187	410	1681	1895

TABLE II
RUNNING TIMES FOR STATIC ANALYSIS OF THE TARGET SWCS

SW Comp.	Frama-C EVA		Slicing	
	Mem. (MB)	Time (s)	LOC (before)	LOC (after)
LCS-C	1281.58	87.96	87340	1469
LCS-S	6564.27	474.04	216349	4923
ApCom	7635.43	596.77	216349	15973
Middleware	1628.26	360.34	106153	16536

properties automatically generated by CBMC (e.g. possible arithmetic overflows, safety of pointer dereferences; see [6]). To enable k-Induction, we instrumented our code base with the necessary assumptions and assertions similarly to Fig. 2. In the k-Induction experiments, we additionally checked constraints on permissible values of variables (e.g., to identify invalid states in the LifeCycle service). Note that defining these latter properties is a manual step that requires insights into the implementation details and the in-depth understanding of the application domain, while the other introduced assertions are automatically constructed.

C. Experiments and Results

For verification we used CBMC 5.23. All experiments were conducted on an Intel(R) Xeon(R) CPU E5345@2.33GHz equipped with 47.2 GB of memory, running Ubuntu 18.04.4. For each run, we set a memory limit of 40 GB and a CPU time limit of one hour, measured by the tool `BenchExec` [16].

1) Static Analysis: We introduced static analysis into our work-flow to address three challenges. First, to avoid spurious counter examples that were due to imprecise value analysis (see for example our k-Induction experiments later in this section). Second, in some of our benchmarks, due to the imprecise value analysis of the function pointers, cycles in the call graph led to non-termination of CBMC. Finally, the computed call graph allows us to identify and exclude code that is not part of the targeted code base, but is still included in the compilation process. The difference in size (lines of codes) before and after slicing unreachable functions in the input file is given Table II. Hence, in our experiments static analysis is an essential preprocessing step that provides valuable benefits.

To gain these benefits, however, an exhaustive static analysis of the code base for each SWC is necessary. Table II presents the running time and memory requirements of this step for each SWC. Note that this analysis includes a precise value analysis for every global variable and function pointer of the code base and removes the unreachable sections of the SWCs.

2) Bounded Model Checking: We considered 5 iterations of the loop calling the `RunnableEntities` of our SWCs (cf. subsect. II-B). As most loops in automotive real-time software are statically bounded, CBMC was able to automatically determine bounds for most other loops. In addition, CBMC can detect whether there exist executions that iterate the loop more often than pretermined by the given bound, which we used to identify loops that needed to be bounded manually (of which there were less than 10 overall).

Table III (left) summarizes our BMC results, providing for each SWC the number of checked assertions, memory usage, and run-time. Though no real bugs were found, our verification attempts revealed a modelling flaw in the ARXML specification of the ApCom SWC. In our first verification attempt, CBMC reported an arithmetic overflow in ApCom. Analyzing the report showed that the ARXML specification of the data type of one of the involved variables (whose value was provided by our ARXML-derived RTE abstraction) was too permissive. As the actual implementation of the RTE is more restrictive, this overflow cannot occur in practice.

We identified a similar problem with the ARXML-derived RTE model of the LCS-C component, which yielded a `Not Present` state that is unreachable in the actual implementation. This revealed a limitation of our modular verification approach, which lacks precise information about the states reachable in other (abstracted) components. As before, this bug cannot occur in the implementation.

The Middleware turned out to be too challenging to verify in our experiments. Attempts to simplify the program (by e.g. abstracting away the initialization of shared memory regions which introduced large arrays in the resulting formulas) led to numerous spurious error reports, rendering the approach impractical. Since CBMC did not support some necessary operations, our attempts to deploy a Satisfiability-Modulo-Theory (SMT) solver as back-end also failed.

3) k-Induction: The right part of Table III presents the results of our k-Induction experiments. The run-times are the sum and the memory requirements are the maximum of the two consecutive CBMC runs for the base case and induction step (see Fig. 2). In our experiments, we observed that a value of 1 is sufficient in all our (terminating) runs to prove the properties, which we attribute to the auxiliary constraints provided by the upfront static analysis. Hence, k-Induction uses fewer resources than BMC in our setting.

Moreover, the value constraints provided by Frama-C proved to be crucial. Our verification attempts without static analysis led to spurious reports of out-of-bound array accesses in the LCS-S component. This is owed to the fact that the initial states (of the state machine) in the induction step (Fig. 2) are arbitrary and hence potentially unreachable in

TABLE III
EXPERIMENTAL RESULTS OF BOUNDED MODEL CHECKING AND k-INDUCTION

SW Comp.	Bounded Model Checking				k-Induction			
	Assertions	Memory (MB)	Time (s)	Outcome	Assertions	Memory (MB)	Time (s)	Outcome
LCS-C	366	1766.5	102.64	Bounded-Success	370	711.6	44.65	Success
LCS-S	1806	2072.2	135.34	Bounded-Success	1824	1334.7	91.04	Success
ApCom	15562	3406.4	157.58	Bounded-Success	15597	3184.0	292.27	Success
Middleware	9680	14635.7	3600.00	Time out	9780	10043.1	3600.0	Time out

the actual implementation. The value set information provided by Frama-C constrains the initial states to reachable states and strengthens our induction hypothesis. Other components (LCS-C and ApCom) could be verified even without the use of Frama-C. As in our BMC experiments, our attempts to verify the Middleware timed out.

For a comparison of (an older version of) CBMC to alternative software model checking tools (such as CPAChecker [17] and Ultimate Automizer [18]) on the presented SWCs, see [7] (Section 6, pages 44-45).

IV. RELATED WORK

Ahmed and Safar [19] use the symbolic simulation tool KLEE [20] to automatically extract test cases from the C source code of an AUTOSAR BSW module. As testing of safety-critical applications must be requirements-based [1], generated test-cases need to be mapped to requirements. In their CBMC-based automated testing method for the avionic domain, Sun et al. [21] annotate the source code with low-level requirements (expressed as pre- and post-conditions) to establish such a mapping. Mittag [22] applies static analysis to AUTOSAR components, focusing on comparatively simple properties. Berger et al. [23] apply the CBMC-based verifier BTC [24] to check automotive code generated by Simulink, but do not address AUTOSAR. Fang et al. [25] use the SPIN model checker to verify a hand-crafted model of an AUTOSAR-based operating system. Westhofen [26] implements custom k-Induction on top of CBMC to efficiently verify automotive C code.

V. DISCUSSION AND CONCLUSION

Automation was a primary goal, as it enables automated regression verification and limits the effort for the verification engineer. The CBMC model checker and its mature ANSI-C support allowed to use our existing build system and largely unmodified code base. The ARXML component descriptions and the layered architecture of AUTOSAR made it possible to delimit the SWCs and automate the generation of a test harness and stubs that abstract the behaviour of the RTE.

We did, however, face challenges regarding automation, modeling the environment, and scalability. Unlike SWCs, CDDs are not standardized by AUTOSAR. They may use interfaces that are not available to standardized SWCs (e.g., to directly access peripherals). Consequently, the stubs for non-standardized interfaces specific to a CDD need to be generated manually. Moroever, even for SWCs, an overly abstract model of the RTE may lead to false positives. This can be addressed by providing a more precise model of the RTE (requiring substantial insight into the details of the RTE) or by including actual RTE code. The latter approach, however, amounts to verifying the SWC in the *absence* of an environment.

As CBMC provides limited support for static analysis, we combined it with an upfront run of Frama-C in order to reduce the computational effort for the model checking – interfacing the tools required a non-trivial implementation effort.

Preliminary experiments showed that verifying multiple, interacting components reduces spurious bug reports. This, however, would require to take into account all execution schedules of the runnables, which we consider future work. Another future work is to reuse our verification efforts of the presented SWCs whenever a repeated analysis is necessary (i.e. when the implementation is changed or re-configured) by considering incremental verification techniques.

Overall, our conclusion and outlook is positive: despite all challenges and the engineering effort required to deploy CBMC to verify AUTOSAR components, we ultimately succeeded in checking non-trivial and realistic SWCs.

ACKNOWLEDGMENTS

This work was partially funded by the Vienna Science and Technology Fund (WWTF) under grant NXT19-006. The authors thank the anonymous reviewers for their valuable feedback and suggestions.

REFERENCES

[1] ISO/TC 22/SC 32, "ISO/DIS 26262 Road vehicles – Functional safety," International Organization for Standardization (ISO), Tech. Rep. 26262, 2018.

[2] W. Steiner, "Formal methods in industrial dependable systems design - the TTTech example," in *Formal Methods in Computer-Aided Design (FMCAD)*, D. Stewart and G. Weissenbacher, Eds. IEEE, 2017, p. 8. [Online]. Available: https://doi.org/10.23919/FMCAD.2017.8102232

[3] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 7, 2008.

[4] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.*, vol. 41, no. 4, 2009.

[5] A. Imparato, R. R. Maietta, S. Scala, and V. Vacca, "A comparative study of static analysis tools for AUTOSAR automotive software components development," in *International Symposium on Software Reliability Engineering (ISSRE) Workshops*. IEEE, 2017.

[6] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, vol. 2988. Springer, 2004.

[7] T. Durand, "Model checking automotive software components," Master's thesis, TU Wien, August 2020.

[8] "Software Component Template - AUTOSAR Rel.4.2.2," Tech. Rep.

[9] "System Template - AUTOSAR Rel.4.2.2," Tech. Rep. [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_TPS_SystemTemplate.pdf

[10] "Specification of RTE - AUTOSAR Rel.4.2.2," Tech. Rep.

[11] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C - A software analysis perspective," in *Software Engineering and Formal Methods (SEFM)*, ser. LNCS, vol. 7504. Springer, 2012.

[12] D. Bühler, "Structuring an Abstract Interpreter through Value and State Abstractions: EVA, an Evolved Value Analysis for Frama-C." Ph.D. dissertation, University of Rennes 1, France, 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01664726

[13] "GitHub eliben/pycparser," https://github.com/eliben/pycparser, accessed: 2021-05-17.

[14] "lxml - XML and HTML with Python," https://lxml.de/, accessed: 2021-05-17.

[15] "GitHub cogu/autosar," https://github.com/cogu/autosar, accessed: 2021-05-17.

[16] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: requirements and solutions," *Int. J. Softw. Tools Technol. Transf.*, vol. 21, no. 1, 2019.

[17] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *Computer Aided Verification (CAV)*, ser. LNCS, vol. 6806. Springer, 2011, pp. 184–190. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_16

[18] M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, and A. Podelski, "Ultimate automizer and the search for perfect interpolants - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, vol. 10806. Springer, 2018, pp. 447–451. [Online]. Available: https://doi.org/10.1007/978-3-319-89963-3_30

[19] M. Ahmed and M. Safar, "Formal verification of AUTOSAR watchdog manager module using symbolic execution," in *International Conference on Microelectronics (ICM)*. IEEE, 2018.

[20] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Operation Systems Design and Implementation (OSDI)*. USENIX Association, 2008.

[21] Y. Sun, M. Brain, D. Kroening, A. Hawthorn, T. Wilson, F. Schanda, F. J. G. Jimenez, S. Daniel, C. Bryan, and I. Broster, "Functional requirements-based automated testing for avionics," in *International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2017.

[22] R. Mittag, "Entwicklung statischer Analysen für AUTOSAR Steuergerätesoftware," Master's thesis, TU Chemnitz, 2018.

[23] P. Berger, J. Katoen, E. Ábrahám, M. T. B. Waez, and T. Rambow, "Verifying auto-generated C code from Simulink - an experience report in the automotive domain," in *Symposium on Formal Methods (FM)*, ser. LNCS, vol. 10951. Springer, 2018.

[24] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller, "Incremental bounded model checking for embedded software," *Formal Aspects Comput.*, vol. 29, no. 5, 2017.

[25] L. Fang, T. Kitamura, T. B. N. Do, and H. Ohsaki, "Formal model-based test for AUTOSAR multicore RTOS," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[26] L. Westhofen, "Verifying automotive C code using modern software model checkers," Master's thesis, RWTH Aachen University, 2019.