International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:13, No:8, 2019

Evaluation of Model-Based Code Generation for Embedded Systems–Mature Approach for Development in Evolution

Nikolay P. Brayanov, Anna V. Stoynova

***Abstract*—**Model-based development approach is gaining more support and acceptance. Its higher abstraction level brings simplification of systems' description that allows domain experts to do their best without particular knowledge in programming. The different levels of simulation support the rapid prototyping, verifying and validating the product even before it exists physically. Nowadays model-based approach is beneficial for modelling of complex embedded systems as well as a generation of code for many different hardware platforms. Moreover, it is possible to be applied in safety-relevant industries like automotive, which brings extra automation of the expensive device certification process and especially in the software qualification. Using it, some companies report about cost savings and quality improvements, but there are others claiming no major changes or even about cost increases. This publication demonstrates the level of maturity and autonomy of model-based approach for code generation. It is based on a real live automotive seat heater (ASH) module, developed using The Mathworks, Inc. tools. The model, created with Simulink, Stateflow and Matlab is used for automatic generation of C code with Embedded Coder. To prove the maturity of the process, Code generation advisor is used for automatic configuration. All additional configuration parameters are set to auto, when applicable, leaving the generation process to function autonomously. As a result of the investigation, the publication compares the quality of generated embedded code and a manually developed one. The measurements show that generally, the code generated by automatic approach is not worse than the manual one. A deeper analysis of the technical parameters enumerates the disadvantages, part of them identified as topics for our future work.

***Keywords*—**Embedded code generation, embedded C code quality, embedded systems, model-based development.

I. Introduction

IN last decades, the automobile has become a complex system, and its main function now is just a small part that is supported by comfort, entertainment, communication and other functionalities. Rising number of the requirements increases the logic and control software for systems in scope and complexity. Millions of lines of code are required, thus meeting the quality and development timeframe requirements is becoming hardly possible.

Model-Based Design is an approach that helps handling this issues. It can be used as a more detailed specification, visualizing the systems interaction and thus simplifying its architecture and behaviour. In the real-time applications, they enable developers to evaluate multiple options, predict system performance, test system functionality by imposing I/O conditions before the product release, and test designs in a virtual environment, thus decreasing the costs, finding defects early in the development cycle and decreasing the total number of defects.

Generally, usage of a common tool environment, exclusion of the manual steps from the process and early defect discovery are expected to bring better performance [1] and decrease of the expenses [2]. The performance of the software depends on models and the code that is generated out of it. In this paper is evaluated how mature are the methods for code generation based on a real use case - ASH. After manual implementation of the model and automatic code generation, the performance of both methods is compared finding if the code generators are good enough to replace the engineers.

Nikolay P. Brayanov and Anna V. Stoynova are with the Technical University of Sofia, bul. "Sveti Kliment Ohridski" 8, 1756 Studentski Kompleks, Sofia (e-mail: npb@ecad.tu-sofia.bg; ava@ecad.tu-sofia.bg).

II. The State of the Art

There are number of terms used to identify software development approaches that focus on models as primary artefacts (in difference to traditional processes that use the source code) [3]. This paper uses model-based software development (MBSD), as it is in connection with the primary aim of the publication – to compare the technical artefacts related to a particular software realisation.

A. Model-Based Software Development - an Approach for Code Generation

The topic of MBSD is complicated and have many different perspectives. The authors [3] enumerate part of the capabilities of this approach, as solution for platform independence, software reuse, creating trustworthy software, architecture analysis of runtime qualities, bridging from problem definition to solution synthesis, implementation of a formal specification, "grand unified theory" of software engineering.

More benefits are enumerated in [4] – a model is easier to maintain and document compared to legacy code and algorithms, features additions and enhancements – shorter time cycles with MBSD, effective as knowledge capture mechanism

Another paper on similar topics [5] discus this question, but none goes in details assessing the generated code's quality, that is at best importance for the systems performance and used hardware, that the price of products depends on. The question for code quality is part of [1], where however they generate

code for a DSP. The results are fascinating, demonstrating hundreds times faster work of generated system. The application of MBSD with microcontroller is investigated [6]. The author compare manually developed code with two versions of generated code – with Embedded Coder and TargetLink. The result shows around 20% difference between sizes of the codes and below 10% difference in execution time. Unfortunately there is no analyses about the reasons of this differences and generally a very small amount of sufficient published researches on the topic.

The quality of the generated code directly depends on the quality of the used model. Formal rules and soft guidelines are helpful for proper model definition. The MAAB Control Algorithm Modelling Guidelines (MathWorks Automotive Advisory Board) [7] is a set of publicly available rules for modelling with Simulink/Stateflow. It has been developed by automotive OEMs and suppliers with the aim to enforce and ease the usage of the MathWorks tools within the automotive industry. The guidelines were published in 2001 and are continuously updated in order to integrate new features. The aim is to achieve system integration without problems, well-defined interfaces, uniform appearance of models, code, and documentation, reusable models, readable models, problem-free exchange of models, a simple, effective process, Professional documentation, understandable presentations, fast software changes, cooperation with subcontractors, successful transitions of research or predevelopment projects to product development. The rules are conceived as a reference baseline.

The most summarized definitions of software quality is given by the IEEE Standard Glossary of Software Engineering Terminology [8], [9]. They define it as “the degree to which a system, component or process meets specified requirements” or “the degree to which a system, component or process meets the needs or expectations of a user”. Going into details discussed in [10], [11], the code quality is relevant to correctness, reliability, efficiency, integrity, usability, maintainability, flexibility, testability, portability, reusability, etc. This paper is focused on the code generation and comparison of its parameters, rather than on the processes that lead to its construction, even though they are closely related. In this sense it investigates the efficiency, measuring the resources that are used – memory and execution time.

B. Comparable Code Quality Parameters

The non-volatile memory contains the code and constants that are referenced by it. It is easily estimated during compilation time.

The volatile memory contains data, including all global and static variables, and stack. The first could be captured during compilation, however task size estimation is not a routine job. Underestimating the maximum stack usage leads to stack overflow and thus system failure, overestimating means wasting valuable memory resources. In [12] is discussed a model based approach for estimation of the maximum stack usage. This is a prove that formal methods for software development could improve the process adding extra checks, however it can’t be used on already developed code. Another publication [13] investigates the worst case stack measurement and enumerates the possibilities. The use case ASH system has two periodic interrupts with small frequency, and measurement of the stack with a debugger could correctly esteem the maximum consumed stack memory.

Measurement of the worst case execution time is very important for safety-critical embedded systems that have hard real-time characteristics. Failure to meet deadlines may be as harmful as producing wrong output or failure. Estimation of execution time is a difficult problem because of the characteristics of real time software and hardware [14]. In current research the measurements are based on hardware supported software monitoring.

C. MBSD Is Not Panacea

MBSD may be more efficient and may have shorter and cheaper development cycles, however it contains possible weaknesses. For example [14] reports that their usage creates dependencies on the tools that manipulate the model and generate code, and introduces risks associated with the continued availability of the tools and compatibility of new releases of the tools. Moreover, it depend on the system domain (business system, command and control system, avionics system, etc.) and the methods used throughout the system lifecycle. These are part of the reasons why [15] suggests that assessment of a tool should be conducted only after gaining an understanding of software engineering methods and choosing a particular method. Thus, comparison of the codes can hardly be stated as objective. To have a representable research it needs to be applied on many different types of functionalities as well as different tool chains.

III. The Use Case and Its Model

An ASH project is used to demonstrate the capability of the code generated from models. The functionality appears in most of the modern cars and is responsible for heating up the seats to required temperature. Commonly, it contains a heating element, temperature sensor, command gateway and a controller that should manage all of them [16]-[18].

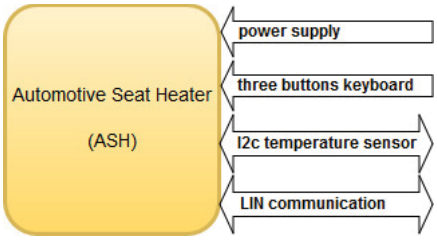

Fig. 1. Automotive seat heater system

In the current case, a LIN communication channel and two buttons are responsible for setting requested temperature. The third button selects if communication or buttons commands should be handled. Additionally, the ECU is responsible for measurement of the supply voltage and behave differently in case of over/under/normal voltage Fig.1.

The use case was chosen as representative, containing

different functionalities, digital and analog signals, sensors and actuators, communication. It has periodical and interrupt services and is simple enough, so all the functionalities could be run in one task for easier measurements.

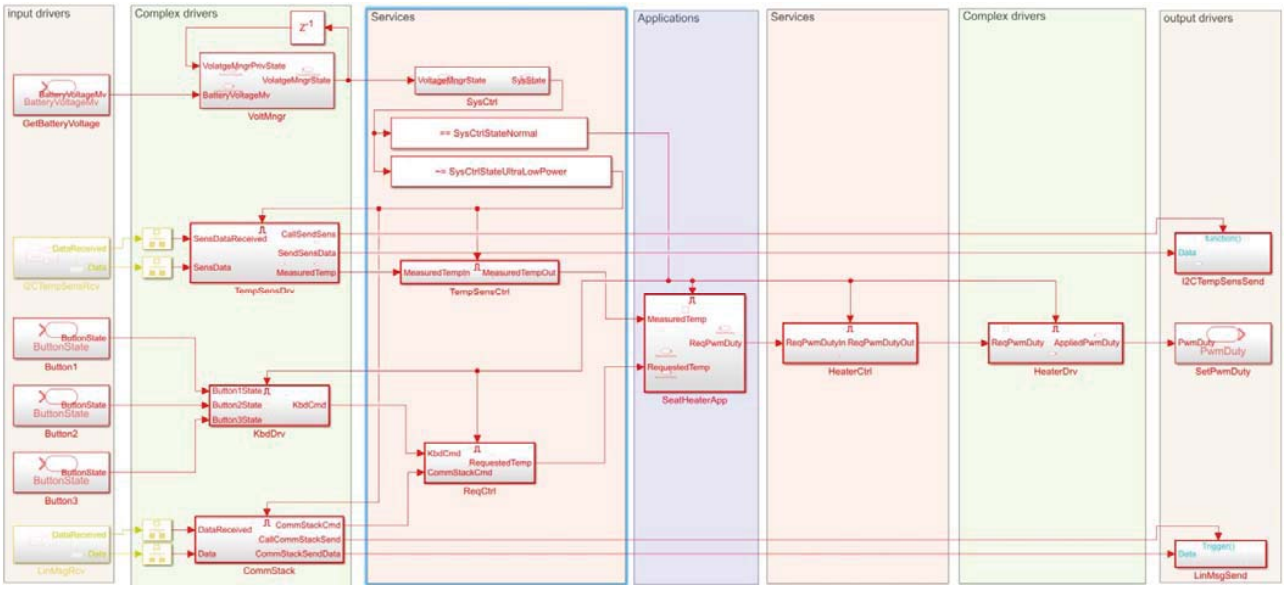

Fig. 2. Model of automotive seat heater

Based on the MAAB criteria was created a model of the device, Fig. 2.

The model architecture contains 4 layers - input and output drivers, complex drivers, services and applications. Embedded coder is used for target code generation, giving priority to ROM and then RAM optimization. Additionally, the model and the architecture that it defines were used for manual software implementation. Both software packages were used for system tests, evaluating their behaviour.

IV. Results

Current paper is evaluating the maturity of code generation, comparing the technical parameters memory usage and execution time for manually developed and generated code, using three compilers so to achieve required level of objectivity.

In Table I are listed all used modules and the memory that they consume. The data is parsed from the particular map files in release mode. It is important to be noted that this results are not representable as comparison between compilers. The configuration of the different compilers is not intended to be equal. As for the fair comparison of compiled code, compilers should use same configuration with manual and generated codes.

The table shows out the code and data memory taken from each particular module compiled with a particular complier. Additionally the difference between generated and manual code based on the size of the manual code is calculated and presented in percent. Thus, having 1% positive difference means that generated code is 1% bigger than the manual one and -1% shows that it is 1% smaller than it. This method is used for all other calculations.

Analysing the used data one could observe enormous differences in the data section – for example 4000% in main, -100% in many others. This multiplies for all three compilers. Looking into the code was concluded that generated code distributes the variables in different way and some of the variables are mapped differently. A further prove for this is found during general memory usage comparison, placed in a later table.

Almost all generated modules use more memory to store their code. The reported difference is from 6 up to 133%, and the last one is remarkable. The analysis shows that in most of the cases, the small differences are caused by the extra checks of interface data that code generator inserts to increase the algorithm's protection. The manual coded variant is based on verified and limited interface data and this kind of checks are considered to be redundant. Additionally, part of the modules are small and even a small code overhead causes a big percentage difference.

The largest difference is reported in SysCtrl and the reason, aside of already mentioned interface data checks, is a tricky code optimization that have been applied. Using tables with function pointers is fast and effective method for optimization of multimodal systems as proposed [19]. It can't be applied on model based bases, however its usage for projects with higher safety integrity level is not allowed, so generally this cannot be assumed as a particular issue.

In contrast of the extra memory used by most of the modules, generated version of HeaterApp consumes 27 to 39%, or 64 to 112 bytes less memory to store the code. The code generator is capable to optimize the calculations, in relevance with available arithmetic. Instead of large types' memory and time-consuming operations as multiplication and division, it uses a sequence of shift operations, thus bringing optimization in code size, but also in execution time, as demonstrated later. The comparisons

of average code sizes are published in Fig. 3 for code and Fig. 4 for data. Each module is represented by its sequence number from the table behind.

TABLE I
CODE AND DATA SIZE IN BYTES, CORRESPONDING TO DIFFERENT COMPILERS

Module	Coding Style	TI v16.9.4.LTS		GNU v6.2.1.16 (SOMNIUM)		IAR C/C++ Compiler for MSP430 7.11.1	
		code	data	code	data	code	data
Main	Manual	386	1	370	1	462	5
	Generated	480	41	496	41	576	44
	Diff, %	24.4	4000.0	34.1	4000.0	24.7	780.0
HeaterApp	Manual	240	12	290	12	214	12
	Generated	176	6	178	6	144	6
	Diff, %	-26.7	-50.0	-38.6	-50.0	-32.7	-50.0
Comm	Manual	292	13	272	13	260	13
	Generated	406	18	456	18	364	18
	Diff, %	39.0	38.5	67.6	38.5	40.0	38.5
HeaterDrv	Manual	236	3	266	3	370	2
	Generated	264	3	318	3	410	2
	Diff, %	11.9	0.0	19.5	0.0	10.8	0.0
KbdDrv	Manual	292	10	366	10	310	9
	Generated	382	0	414	0	362	0
	Diff, %	30.8	-100.0	13.1	-100.0	16.8	-100.0
TempSensD	Manual	400	30	484	30	498	32
	Generated	426	30	534	30	540	32
	Diff, %	6.5	0.0	10.3	0.0	8.4	0.0
VoltMngr	Manual	286	6	362	6	380	3
	Generated	270	1	344	1	298	1
	Diff, %	-5.6	-83.3	-5.0	-83.3	-21.6	-66.7
HeaterCtrl	Manual	32	2	42	2	32	2
	Generated	40	0	48	0	38	0
	Diff, %	25.0	-100.0	14.3	-100.0	18.8	-100.0
ReqCtrl	Manual	132	8	148	8	130	6
	Generated	166	0	206	0	156	0
	Diff, %	25.8	-100.0	39.2	-100.0	20.0	-100.0
SysCtrl	Manual	150	26	174	26	134	12
	Generated	350	4	406	4	280	3
	Diff, %	133.3	-84.6	133.3	-84.6	109.0	-75.0
TempSensC	Manual	22	2	28	2	22	2
	Generated	26	0	40	0	30	0
	Diff, %	18.2	-100.0	42.9	-100.0	36.4	-100.0

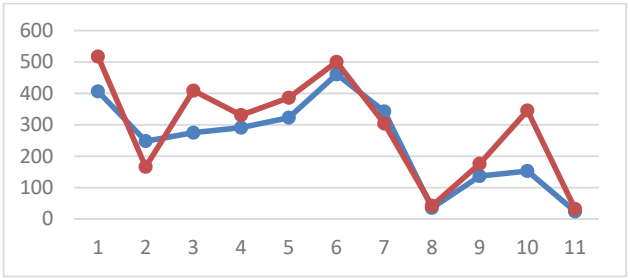

Fig.3. Average code size of the used modules, represented by its sequence number. Generated – red; manual – blue

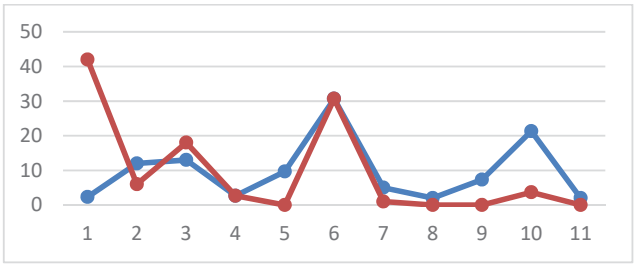

Fig.4. Average code size of the used modules, represented by its sequence number. Generated – red; manual – blue;

To calculate the overall memory usage, the required stack size should be added Table II.

TABLE II
MEASURED USAGE OF STACK

Coding Style	TI v16.9.4.LTS	GNU v6.2.1.16 (SOMNIUM)	IAR C/C++ Compiler for MSP430 7.11.1	Average
Manual	61	42	44	**49**
Generated	75	63	57	**65**
Diff	**14**	**21**	**13**	**16**
Diff, %	**23.0**	**50.0**	**29.5**	**32.7**

The experiment shows that generated code uses 14 to 21 bytes more, that is 23 to 50%. As already shown, the generated code consumes less data than manual one, but the stack size is bigger. It comes to be a difference between the particular manual coding style and the generated one. Summarized data from ROM and RAM usage is in table 3.

Execution times have enormous deviations up to 130%. The execution times for generated HeaterApp is 80% less than the manual implementation because the already mentioned optimization of complex calculation. The deviations in small measurements are bigger because of the small overheads caused by measurement approach as well as the fact that a small difference represents a large percent from the benchmark value. What looks to be unreasonable is the 103% difference in KbdDrv. This relevantly simple module interacts with many others and so measuring its execution time includes these calls. Thus, individual calculations are deemed to not correctly represent the parameters and summarized data are suggested as indicative for the systems' performance.

TABLE III
AUTOMOTIVE SEAT HEATER OVERALL MEMORY USAG

Coding Style	Memory type	TI v16.9.4.LTS	GNU v6.2.1.16 (SOMNIUM)	IAR C/C++ Compiler for MSP430 7.11.1	Average
Manual	code+const	4928	5333	5308	**5190**
	data+stack	182	163	150	**165**
Generated	code+const	5446	5971	5694	**5704**
	data+stack	186	174	171	**177**
Diff	code+const	**518**	**638**	**386**	**514**
	data+stack	**4**	**11**	**21**	**12**
Diff, %	code+const	**10.5**	**12.0**	**7.3**	**9.9**
	data+stack	**2.2**	**6.7**	**14.0**	**7.3**

Measuring the execution times result in Table IV.

TABLE IV
AUTOMOTIVE SEAT HEATER WORST CASE EXECUTION TIMES

Modul	Coding Style	TI v16.9.4.LTS	GNU v6.2.1.16 (SOMNIUM)	IAR C/C++ Compiler for MSP430 7.11.1
Heater App	**Manual**	33.7	28.3	23.8
	Generated	6.7	6.0	5.6
	Diff, %	-80%	-79%	-76%
Comm Stack	**Manual**	10.8	10.6	10.6
	Generated	11	10.9	10.8
	Diff, %	2%	3%	2%
Heater Drv	**Manual**	37.2	23.9	12.2
	Generated	37.7	24.3	12.4
	Diff, %	1%	2%	2%
KbdDrv	**Manual**	6.3	7.7	8.7
	Generated	12.8	12.3	10.6
	Diff, %	103%	60%	22%
Temp SensDrv	**Manual**	5.5	6.1	6.5
	Generated	5.6	6.1	6.5
	Diff, %	2%	0%	0%
Volt Mngr	**Manual**	34.2	28.9	24.6
	Generated	24.8	24.5	24.3
	Diff, %	-27%	-15%	-1%
Heater Ctrl	**Manual**	4.2	4.9	5.3
	Generated	5.3	4.2	3.5
	Diff, %	26%	-14%	-34%
ReqCtrl	**Manual**	5.3	5.2	5.2
	Generated	3.8	4.7	5.3
	Diff, %	-28%	-10%	2%
SysCtrl	**Manual**	10.0	9.6	9.4
	Generated	8.6	10.2	11.4
	Diff, %	-14%	6%	21%
Temp SensCtrl	**Manual**	4.2	4.9	5.3
	Generated	5.3	5.2	5.2
	Diff, %	26%	6%	-2%

Fig. 5 visualize the worst-case execution time for each of the modules. It is calculated as average of the measurements of all compilation versions.

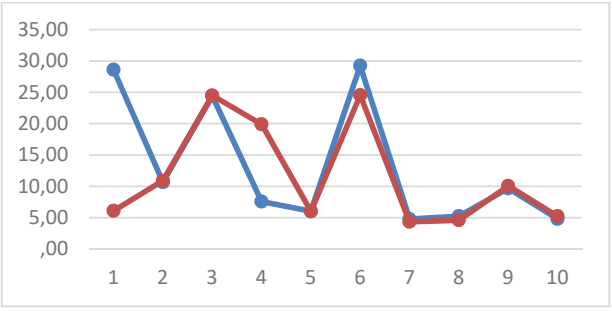

Fig.5. Average worst case execution time of each module, represented by its sequence. Generated – red; manual – blue

To summarize, all worst-case times are summed, estimating the worst-case system's behaviour. Additionally, measurement of execution time of the real integrated system is done. Results are part of Table V.

TABLE V
SUMMARIZED EXECUTION TIMES – A SUM OF ALL MODULES WORST CASES AND THE FULL INTEGRATED SYSTEM'S WORST CASE

		TI v16.9.4.LTS	GNU v6.2.1.16 (SOMNIUM)	IAR C/C++ Compiler for MSP430 7.11.1	average
SUM	**Manual**	151.4	130.1	111.6	**131.0**
	Generated	121.6	108.4	95.6	**108.5**
	Diff, %	**-19.7**	**-16.7**	**-14.3**	**-17.2**
Whole task	**Manual**	119.7	95.3	77.8	**97.6**
	Generated	110.5	82.2	63.3	**85.3**
	Diff, %	**-7.7**	**-13.7**	**-18.6**	**-12.6**

The summed up times are greater than the whole system time, because of measurement overhead and the fact that in the real live is impossible all modules to work on maximal load in same time. For the summed performance are calculated 17% difference. When it is measured in real time work the manual implementation takes up to 13% more time.

V. CONCLUSION

In current research was evaluated the code generation for embedded systems, using Matlab tools and ASH as a use case. The comparison between generated and manually created implementation shows that generated code consumes 10% more non-volatile memory and 7% more volatile memory.

The reason for this difference in code's size was found in extra check that is inserted to validate modules' interface data. Additionally, manually developed code use an optimization, based on function pointers, that model based approach does not support. This is a minor benefit, since this technic is not allowed for higher safety integrity levels.

Volatile memory consists of data and stack. The size of used data by both of approaches is very similar, and generated code consumes only 3% less memory (4 bytes). Stack usage, however differs and generated code consumes up to 50% more memory, depending on the used compiler. This shows that automatic coding style is more stack-hungry.

Measured execution times for different modules varies up to 100%. It was noticed that automatic code generation is capable to optimize complex calculations using a simple bit shifting. This makes the generated code running 13% faster on average.

The MBSD is a complex approach based on abstractions and depending on the technical as well as non-technical context, thus general conclusion about its applicability is impossible. However, based on particular case was proved that the generated code has relevantly small deviations from the manually developed and this makes is suitable and productive because of all other benefices that the method carries.

ACKNOWLEDGEMENT

The paper is published with the support of the project No BG05M2OP001-2.009-0033 "Promotion of Contemporary Research Through Creation of Scientific and Innovative Environment to Encourage Young Researchers in Technical University - Sofia and The National Railway Infrastructure Company in The Field of Engineering Science and Technology Development" within the Intelligent Growth Science and Education Operational Programme co-funded by the European Structural and Investment Funds of the European Union.

REFERENCES

[1] Einfochips, "MODEL-BASED DESIGN FOR EMBEDDED SOFTWARE", White Paper, https://www.einfochips.com/wp-content/uploads/resources/model-based-design-whitepaper.pdf
[2] J. Lin, "Measuring Return on Investment of Model-Based Design", White Paper,https://www.mathworks.com/content/dam/mathworks/mathworks-dot-com/solutions/model-based-design/mbd-roi-video/Measuring_ROI_of_MBD.pdf
[3] J. Klein, H. Levinson, J. Marchetti, "Model-Driven Engineering: Automatic Code Generation and Beyond", CMU/SEI-2015-TN-005
[4] B. Schatz, A. Pretschner, F. Huber, J. Philipps, "Model-Based Development of Embedded Systems", DOI: 10.1007/3-540-46105-1_34
[5] M. Broy, S. Kirstan, H. Krcmar, B. Schätz, J. Zimmermann, „What is the benefit of a model-based design of embedded software systems in the car industry?", DOI: 10.4018/978-1-4666-4301-7.ch017
[6] N. Ajwad, "Evaluation of Automatic Code Generation Tools", ISSN 0280-5316
[7] MathWorks® Automotive Advisory Board "Control Algorithm Modeling Guidelines Using MATLAB®, Simulink®, and Stateflow", https://www.ethz.ch/content/dam/ethz/special-interest/mavt/dynamic-systems-n-control/idsc-dam/Lectures/Embedded-Control-Systems/AdditionalMaterial/Miscellaneous/MAAB%20Control%20Algorithm%20Modeling%20Guidelines%20using%20MATLAB%20Simulink%20and%20Stateflow.pdf
[8] "610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology", DOI: 10.1109/IEEESTD.1990.101064
[9] "1219-1998 - IEEE Standard for Software Maintenance", 10.1109/IEEESTD.1998.88278
[10] S. H. Kan, "Metrics and Models in Software Quality Engineering, Second Edition", ISBN: 0-201-72915-6, Chapter 4: Software Quality Metrics Overview
[11] D. Chappell, "THE THREE ASPECTS OF SOFTWARE QUALITY: FUNCTIONAL, STRUCTURAL, AND PROCESS", http://www.davidchappell.com/writing/white_papers/The_Three_Aspects_of_Software_Quality_v1.0-Chappell.pdf
[12] C. Ferdinand, R. Heckmann, H. J. Wolff, C. Renz, O. Parshin, R. Wilhelm, "Towards Model-Driven Development of Hard Real-Time Systems Integrating ASCET and aiT/StackAnalyze", https://www.absint.com/aiT_ASCET.pdf
[13] T. Yu, M. B. Cohen, "Guided Test Generation for Finding Worst-Case Stack Usage in Embedded Systems", DOI: 10.1109/ICST.2015.7102592
[14] S. Petters, "Comparison of Trace Generation Methods for Measurement Based WCET Analysis", In Proceedings of the 3rd International Workshop on Worst Case Execution Time Analysis, 2003, page 61-64
[15] R. Firth, V. Mosley, R. Pethia, L. Roberts, W. Wood, "A Guide to the Classification and Assessment of Software Engineering Tools", CMU/SEI-87-TR-010
[16] "Heated Seat Module", http://moojohn.com/truck/heatedseats2.pdf
[17] "Car seat heater", https://www.autokraitis.lt/documents/csh_en___print_v1.pdf
[18] "Heated seat system", http://moojohn.com/truck/heatedseats.pdf]
[19] A. Milanova, A. Rountev, B. G. Ryder, "Precise Call Graphs for C Programs with Function Pointers", Automated Software Engineering (2004) 11: 7. https://doi.org/10.1023/B:AUSE.0000008666.56394.a1